Client/Server

Avoiding data-races by using the client/server pattern

Concurrency pattern: client/server

- This pattern is used sometimes to avoid data-races when you have a shared resource.
- Associate a single thread to operate the resource. This thread is called the *server*.
- The threads requiring the resource are called the *clients*. They don' t have access to the resource.
- Clients send *messages* to the server for requesting operations requiring the resource.

Example: Printing documents

- An object of the Printer class is able to print a string on a printer device.
- Create the object:

```
Printer prt= new Printer("/dev/printer0");
```

- Print a multi-line string with:

```
String text= "...";
prt.print(text);
```

- The object `prt` does not support concurrent invocations.

A printer server

- The problem consist of programming a class PrintServer with support for concurrent calls.

```
static PrintServer server= new PrinterServer("...");
...
// several threads execute:
String text= ...;
long millis= server.print(text);
System.out.println("elapsed time= "+millis);
```

- Trivial solution: use synchronized.

Asynchronous requests

- Aditionnal constraint: clients can submit asynchronous requests.

```
String text= ...;
PrintRequest req= server.submitPrint(text);
... do another thing ...
long millis= server.waitPrint(req);
```

- Clients can execute other activities while waiting for a request.
- Cannot be solved using just synchronized.

An example of a simple inter-thread message system

- The clients communicate with the server by exchanging messages.
- class Port:

```
void send(Message msg); // sync
void submit(Message msg); // async
void waitReply(Message msg);
Message receive();
void reply(Message msg);
void forward(Message msg, Port port);
```

- class Message:

```
Thread getSender();
```

Printer server implementation

```
class PrintRequest
      extends Message {
  String text;
  long millis;
  PrintRequest(String text) {
    this.text= text;
  }
}

class PrinterServer
      extends Port
      implements Runnable {
  Printer prt;
  PrinterServer(String dev) {
    prt= new Printer(dev);
    new Thread(this).start();
  }

  // The asynchronous part
  ...
```

```
long print(String text) {
  PrintRequest req= new
       PrintRequest(text);
  send(req);
  return req.millis;
}

// this is a private method
public void run() {
  for(;;) {
    PrintRequest req=
     (PrintRequest)receive();
    long ini=
        System.currentTimeMillis();
    prt.print(req.text);
    req.millis=
        System.currentTimeMillis()-
        ini;
    reply(req);
} }
```

Asynchronous part

```
PrintRequest submitPrint(String text) {
  PrintRequest req= new PrintRequest(text);
  submit(req);  // Don't wait for printing
  return req;
}

long waitPrint(PrintRequest req) {
  waitReply(req);  // Wait if not still printed
  return req.millis;
}
```

Generalization

```
class SomeRequest
  extends Message {
  parameters
  outputs
  constructor
}
class SomeServer
      extends Port
      implements Runnable {
  ...
  type someService(...) {
    SomeRequest req= new ...;
    send(req);
    return ...
  }
```

```
  SomeRequest
      submitService(...){ ... }
  type waitService(...) { ... }

  SomeServer(...) {
    ...
    new Thread(this).start();
  }
  public void run() {
    for (;;) {
      ... initialization ...
      SomeRequest req=
          (SomeRequest)receive();
      ...
} } }
```

Messages as a synchronization mechanism

- Messages can also be used to implements synchronization policies.
- For example: a semaphore.
- waitTicket and signalTicket are implemented as synchronous messages to a server thread.
- To avoid that waitTicket be passed when there are no tickets, the server thread does not reply them.

```
public class Semaphore
     extends Port
     implements Runnable {
  int initial;
  public Semaphore(
           int initial) {
    this.initial= initial;
    new Thread(this)
      .start();
  }

  static class WaitMsg
         extends Message { }
  public
  void waitTicket() {
    send(new WaitMsg());
  }

  static class SignalMsg
         extends Message { }
  public
  void signalTicket() {
    send(new SignalMsg());
  }
```

```
  public void run() {
    int tickets= initial;
    List queueList=
              new LinkedList();
    for (;;) {
      Message msg= receive();
      if (msg instanceof
              SignalMsg) {
        reply(msg);
        if (queueList.isEmpty())
          tickets++;
        else {
          Message waitMsg=
             (Message)queueList
                .remove(0);
          reply(waitMsg);
        }
      }
      else {
        if (tickets<=0)
          queueList.add(msg);
        else {
          tickets--;
          reply(msg);
} } } } }
```

Extensions

- Multiple resources controlled by a single server:
  - Associate a thread to each resource.
  - A single controller thread receives all request messages.
  - The controller forwards the messages to the resource threads.
- Distributed implementations:
  - Run the threads on processors with non shared memory.

Summary

- Concurrency patterns: client/server.
- Concepts: asynchronous requests.
- Synchronization mechanisms: message systems.