Project 4
CMSC 433
Programming Language Technologies and Paradigms
Fall 2002

Due Wednesday, November 27, 2002 at 6pm

Introduction

The goals of this project are to develop more experience with concurrency, to learn about asynchronous event handling, and to explore reconfiguring applications at runtime.

The project will involve extending our previous server applications with a monitoring subsystem. The server will log events of interest. These events will be monitored by the monitoring subsystem and, under certain conditions, the monitor will direct the server to reconfiguring itself in order to improve the server’s performance.

1 ReconfigPageServer

ReconfigPageServer is an application composed of a (slightly modified) BoundedThreadServer, some handlers from project 3, and a new handler, called ReconfigHandler. This handler will process requests that reconfigure the server in some way. In this projects, reconfigurations can have two flavors: changing the handler list (adding or removing a handler), or changing the number of processing threads.

1.1 Modified BoundedThreadServer

To support adding and removing handlers, we must change the signature of the BoundedThreadServer to be the following:

```java
public class BoundedThreadedServer {
    public BoundedThreadedServer(int port, int maxThreads);
    public void attach (HttpHandler handler, String handlerName)
        throws HandlerNameInUseException;
    public void detach (HttpHandler handler);
    public HttpHandler getHandlerByName (String handlerName);
    public void adjustMaxThreads(int k);
    public void run ();
}
```
Here is a summary of the changes:

- The attach method now takes an additional String argument, which is the name of the handler. The server must keep track of which handler has which name. Attach() should fail (throwing HandlerNameInUseException) if a handler called handlerName is already attached to the server.

- getHandlerByName returns the HttpHandler with name handlerName if it has been attached. It returns null otherwise.

- Finally, adjustMaxThreads adjusts the count of the maximum number of threads that the server can run at a given time by k. If the adjustment would cause the maximum number of threads to be less than or equal to 0, then it is set the maximum number to 1.

1.2 ReconfigHandler

Here is the specification for this handler:

- ReconfigHandler. Matches “reconfig/*” and is exclusive. The ReconfigHandler constructor has the following form

  public ReconfigHandler(HandlerOrder orderConstraint, 
                          BoundedThreadedServer server)

  The order constraint is as usual, and the BoundedThreadedServer is the server to which the ReconfigHandler is attached.

  Client requests are of the following forms:

  - http://Server:Port/reconfig/adjustMaxThreads=x

    The handler extracts x from the request and adjusts the maximum number of threads for its server by x.

  - http://Server:Port/reconfig/detach=name

    The handler detaches the handler named name from its BoundedThreadedServer. It keeps track of the returned HttpHandler object and that object’s name in order to implement the third kind of request, below.

  - http://Server:Port/reconfig/reattach=name

    This indicates that the handler should reattach a previously detached HttpHandler (using the detach URL above) having name name.

Be aware that you will need to deal with shared data when implementing this handler, so take care to use proper synchronization.
1.3 Output of EndTimerHandler and EndSessionHandler

To facilitate better testing, please ensure that your EndTimerHandler outputs its results to the log in the following format:

Request requestName processed in time ms

and EndSessionHandler should output records of the form:

Session key ended; elapsed time: time ms

1.4 The SimpleReconfigPageServer Application

To build the application, in the main method of the class SimpleReconfigPageServer you will create the handlers as follows; unless otherwise noted, they should have order constraint NO_PREFERENCE:

1. Create the same handlers (i.e. RemoteGetFileHandler, StartTimerHandler, SummaryHandler, etc.) that you did in your SeqPageProxy in Project 3. For this application, StartTimerHandler and EndTimerHandler will write their LogRecords to a LocalLog; this part will change in Section 3.

2. In addition, create a ReconfigHandler, passing it an instance of a BoundedThreadServer, call it server.

3. Attach all created handlers to server, and start it running.

   All applications should accept the -Dport=XXX flag to set the port to use to listen for connections. In addition, all of the names of the handlers that you attach should be exactly the name of their class; for example:

   BoundedThreadedServer bth = new BoundedThreadedServer(port,num);
   HttpHandler eth = new EndTimerHandler(t,p,1,c);
   bth.attach(eth,"EndTimerHandler");
   ...

   Finally, be sure to accept the -Dnumthreads=XXX flag for setting the number of threads.

2 RemoteLogMonitor

The second part is to generalize RemoteLogServer from project 1: rather than only processing messages it receives on a single LocalLog, a RemoteLogMonitor will now, in addition, also forward the same message to each of several event queues. These queues will be used by other classes discussed later.

   RemoteLogMonitor will have the following signature:
public class RemoteLogMonitor
    Log myLog;
    public RemoteLogMonitor(int port, Log myLog);
    public void attach(EventQ q);
    public void detach(EventQ q);
    public void process() throws IOException;
}

Briefly, the methods work as follows:

- The RemoteLogMonitor constructor takes as its argument the port to
  listen on for messages from a RemoteLogClient. It doesn’t start listening
  yet.

- The attach and detach methods behave as they do in BasicServer; when
  I attach/detach an EventQ, it is added/removed to a list maintained by
  the RemoteLogMonitor.

- The process method starts the server running, waiting for RemoteLog-
  Client messages on the given port.

Each time the RemoteLogMonitor receives a message, it will invoke the
 corresponding method on myLog. It will also create a LogEvent (See below)
 corresponding to the message and enqueue it in each of its attached EventQs.

Once you have implemented this part of the RemoteLogMonitor, you could
 implement what amounts to a regular RemoteLogServer by adding a main
 method to RemoteLogMonitor as follows (you should do this for testing):

- Create an instance of RemoteLogMonitor with an appropriate port (ac-
  quired by checking the -Dport=XXX flag) and an appropriate Log (for
  example, a size 100 LocalLog).

- Start processing messages by calling process.

2.1 Event Queues

One disadvantage of the original RemoteLogServer is that each request must
 be serviced immediately. If this is fast, then this isn’t a problem. However, if
 servicing is slow, it will hold up the processing of other incoming log messages,
 just as our sequential BasicServer did. For example, if myLog is a RemoteLog-
 Client whose server was located somewhere on Pluto, then everything would be
 slowed significantly.

To fix this problem, we can introduce asynchroney via events. In particular,
 we will translate messages into LogEvents which we store in a queue. Another
 thread can then drain the queue to process the events. This other thread can
 run in parallel with the main server thread, and so incoming messages are not
 delayed in their processing.

We can do this in several stages. First, you should create a generic class for
 implementing events of interest to Logs. Second, you should create a a queue in
which to store those events. Finally, you will need to create another class that
dequeues the events and processes them. The first class will be called LogEvent,
the second will be called EventQ, and the third is called EventThread.

2.2 LogEvent
To record events, we’ll create an abstract LogEvent class and subclass it for
every event that an EventThread may need to process.

```java
public abstract class LogEvent {
    public LogEvent();
}

public class AddEvent extends LogEvent {
    public AddEvent(LogRecord r);
    public LogRecord getLogRecord();
}

public class GetAllEvent extends LogEvent {
    public GetAllEvent(long ms);
    public long getWindow();
}

public class SetFilterEvent extends LogEvent {
    public SetFilterEvent(String filter);
    public String getFilter();
}
```

2.3 EventQ
Now that we have a way to create events, we can create a queue to hold them.

```java
public class EventQ {
    public EventQ();
    public void enqueue(LogEvent event);
    public LogEvent dequeue(); // blocks when empty
}
```

The constructor creates the queue (having essentially unbounded size). The
enqueue method stores the given LogEvent in the queue, while dequeue will
return the first element of the queue (and block if the queue is empty).
You must take care to either synchronize your EventQ, or else synchronize
all threads that use it.

2.4 EventThread
To process requests queued on an EventQ, we can use the class EventThread:

```java
public class EventThread extends Thread {
    public EventThread(EventQ eventQ, LogEventHandler handler);
```
public void run();
}
interface LogEventHandler {
    public void process (LogEvent event);
}

The thread should dequeue events from the EventQ, and then process them
by calling the process() method of the provided LogEventHandler. We will
implement two LogEventHandler, one for printing out events to the console,
and another for reconfiguring the remote server.

2.4.1 Printer
To better use the new functionality afforded by RemoteLogMonitor, we create
an EventThread with an implementation of LogEventHandler called Printer
that simply prints to the console (using System.out) what log messages have
been sent to the server. Its function will be to simply print out the fact that
a certain method was called. For example, if an add method was called with
LogRecord l, then Printer will print to the console the word “add” followed by
the string representation of the record l.

You can now create a “printing” version of a RemoteLogServer by creating
a RemoteLogMonitor as described above, but additionally start an instance of
and EventThread with a Printer and an EventQ that has been attached to the
RemoteLogServer.

3 AdvancedReconfigPageServer and ReconfigMonitor
Now that you have a reconfigurable server and a generalized remote logging
monitor, we can make the two interact with each other. In particular, we can
have the monitor watch relevant events being logged by the server, and then
reconfigure the server.

The AdvancedReconfigPageServer application will be a variant of the Ba-
sicReconfigPageServer that talks to a RemoteLogMonitor. In Section 1.4, we
created the BasicReconfigPageServer using the same handlers as SeqPageProxy.
Now however, when we create a Log for use by StartTimerHandler and End-
TimerHandler, and to store in the Map for SummaryHandler, we will create
a RemoteLogClient rather than a LocalLog. The port to use for this client should
be retrieved from the command line, using -Dlogport=XXX.

In addition, you will create a class ReconfigMonitor as follows. The main
method of this class will create an instance of RemoteLogMonitor, and then
attach to it two eventQs. One EventQ will be read by a Printer EventThread
and the other by a new handler, a PerformanceMonitor EventThread. This is
depicted in Figure 1. The PerformanceMonitor will monitor the “health” of
the AdvancedReconfigPageServer, and potentially reconfigure it; we describe
Figure 1: Monitoring and Reconfiguration Architecture
it more below. Once you have attached the eventQ’s, you can start running the ReconfigMonitor by invoking the RemoteLogMonitor’s process() method. Finally, use the -Dwebport=XXX flag for setting the port that your AdvancedReconfigPageServer is listening on.

3.1 PerformanceMonitor

Ideally, the basic idea is that the PerformanceMonitor will analyze the LogEvents it sees, decide whether the server is performing well, and, if it is not, send reconfiguration requests to the server in order to improve its performance. The processing functionality should be placed in a class called PerformanceMonitor that implements LogEventHandler.

For this project you will examine the result of each AddLogEvent generated by EndTimerHandler to decide whether to detach or reattach handlers, or to change the thread threshold. (Recall that the AddLogEvent will be created and stored in the queue by RemoteLogMonitor whenever an EndTimerHandler adds a record to the remote log. You should only process AddLogEvents resulting from an EndTimerHandler, and not other LogEvents.) You will need to continuously calculate a rolling average of the last three EndTimerHandler timing results. Then apply the following rules:

- For the first 50 AddLogEvent records don’t do anything. After that apply the following rules after each new AddLogEvent is processed.

- If the rolling average increases at least 3 out of 5 times, cut down the maximum number of threads by 2.

  For example, say the average starts of as 10.0, and then five messages come in, changing it in the following sequence: 10.1, 9.8, 10.0, 9.7, 9.8. In this case, we cut down the maximum number of threads because the average went up three times over the five messages (from 10.0 to 10.1, from 9.8 to 10.0 and from 9.7 to 9.8). If the next message causes the average to go up to 9.9, then we immediately change the maximum number of threads again, because over the last five messages, the average still went up three times (from 9.8 to 10.0, from 9.7 to 9.8, and from 9.8 to 9.9). If the next message causes the average to drop to 9.5, then we do not want to increase the max threads, because the average now has only gone up 2 times out of the last 5 (from 9.7 to 9.8, and from 9.8 to 9.9).

- If the rolling average increases at least 4 out of 5 times, detach the RemoteGetFileHandler.

- if the rolling average decreases at least 3 times out of 5, increase the maximum number of threads by 2.

- If the rolling average decreases at least 4 out of 5 times, and you’ve detached the RemoteGetFileHandler, then reattach it.
4 Testing

Write a JUnit Test Case class for each of the following classes, with each Test Case class implementing at least two tests:

- (The modified) BoundedThreadedServer
- ReconfigHandler
- RemoteLogMonitor
- Printer
- EventThread
- EventQ
- PerformanceMonitor

Each Test Case class must be self-contained, and runnable from the command line. If you must fork other processes to run your test, do it from with the test class itself (and be sure to kill the processes when you’re done).

5 Amendments

- Cleared up description of eventQ (incorrectly said that enqueue would be generating LogEvents, rather than just enqueuing them).

- Some command-line arguments you need: ReconfigMonitor should accept the `-Dwebport=XXX` flag for setting the port that your AdvancedReconfigPageServer is listening on. Each of the ReconfigPageServers should also accept the `-Dnumthreads=XXX` flag for setting the starting maximum number of threads.

- The descriptions of policies for PerformanceMonitor imply that at least the required condition must occur. That is, if we say the rolling average increases 4 out of 5 times then do X, then you should also do X if it increases 5 out of 5 times.

- Made clarifications about class names to use: for part one, your server is called SimpleReconfigPageServer, and for the third part it's called AdvancedReconfigPageServer. For the third part, you create a class called ReconfigMonitor that creates a RemoteLogMonitor instance for its use.

- Added clarifications about how PerformanceMonitor is supposed to work.

- Indicated that the exact names of handlers SimpleReconfigPageServer and AdvancedReconfigPageServer.

- Make sure the output of your EndSessionHandler and EndTimerHandler is as described in Section 1.3.
• Clarified the job of the PerformanceMonitor.