JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 6, November-December 2003

Branch and Bound Implementations for the Traveling Salesperson Problem -
Part 4: Distributed processing solution using RMI

Richard Wiener, Editor-in-Chief, JOT, Associate Professor, Department of Computer Science, University of Colorado at Colorado Springs

The multi-threaded implementation presented in the previous column (July/August, 2003) sets the stage for the distributed processing implementation to be presented in this column.

Please see the previous two columns for the details regarding the TSP best-first branch and bound algorithm that forms the basis for the work that shall be described in this column.

Class *Node* is unchanged from the single and multi-threaded implementation presented in Part 2 of this series.

Before the mechanics (RMI in this case) of distributed processing can be deployed the algorithm must be setup to support parallel computation. This was accomplished during the multi-threaded design and implementation presented in the previous column. Recall that threads were spawned from an instance of class *TSP* after all the nodes at level 2 were generated. These nodes represent partial tours of size 2 (e.g. [1, 2], [1, 3], [1, 4], …, [1, n]. The nodes were inserted into a priority queue implemented using Java's standard collection class *TreeSet*. These nodes are prioritized in the *TreeSet* according to the value of their computed lower bound (ties being resolved by the sum of the cities in the tour – see the previous column for details).

Instead of spawning threads and handing each thread one of the level 2 nodes, we define class *TSP* as the server and allow clients, defined by the revised class *ProcessNodes* (previously the thread class) to request nodes from the server. These nodes are handed off to the requesting clients until no more nodes are left.

Cite this column as follows: Richard Wiener: "Branch and Bound Implementations for the Traveling Salesperson Problem – Part 4", in *Journal of Object Technology*, vol. 2, no. 6, September-October 2003, pp. 51-65. http://www.jot.fm/issues/issue_2003_11/column5

1 DESIGN CONSIDERATIONS

It is well known that communication between processes across a network is expensive. It is therefore imperative that measures be taken in the design of the distributed application to minimize such inter-process communication.

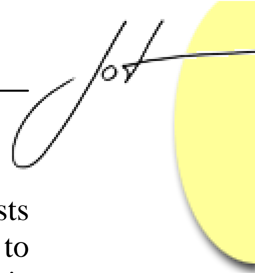
A natural (but not efficient) design approach would be to allow each client to play the dual role of a server in order to allow the server class (class TSP) that doles out nodes to each process when it becomes available to communicate with the client. The need for the server to communicate with each client occurs whenever a client finds a tour whose cost is less than the current best tour. Each of the other client processes needs to have this information available.

Before ceding to the temptation to make each client available to the server (and thus act like a local server as well), an attempt was made to allow only one way communication – from each client to the real server. This in principal should simplify the design and avoid excess inter-process communication. The attempt succeeded after making a few compromises in the design. This design is described below.

When the server starts it generates the root node and then all the level 2 child nodes that must be processed in order to complete the computation. It inserts these nodes into its priority queue (TreeSet).

When each client process running on an independent computer is started, it sends a request to the server for the first available (highest priority) node in the priority queue of nodes residing on the server. It also queries the server for the current best tour value. After a pre-determined number of nodes have been generated (specified by a constant in client class *ProcessNodes* -50,000 in this case), the server is queried for the best tour. Each time a client finds a tour that is better than the best tour value that it knows about, it sends the server a message with this new best tour value (and the node that represents the tour with this best tour value). The server updates its best tour value if the new best tour sent by the client is actually smaller than its currently recorded value of the best tour. The other clients become informed of this new best tour value after they have generated their requisite 50,000 nodes. So the compromise being designed here is that client processes may have slightly stale values for the current best tour. Only the server is up-to-date. Given how relatively infrequently new best tours are found, particularly after the initial state of computation has passed, this compromise is deemed to be a reasonable one. It allows for a great simplification of the design. The server does not need to know about the existence of any of the clients. New client machines may be added to the mix at any time, even after the computation is underway.

As clients complete the processing of their nodes and request a new node from the server, the pool of available nodes eventually becomes depleted in the server. When this server node pool becomes empty the application stops after each client has completed the processing of its final node. This can lead to inefficiency if one or more of the client



machines is (are) significantly slower than the other machines (this very situation exists in the author's network of five machines). Then the slowest machine may continue to process its final node long after the fastest machines have completed their work. This performance penalty may in fact offset the benefit obtained by having the slow machine participate in the distributed processing.

To cope with this problem of uneven load balancing during the end game (the processing of the final nodes by each processor), a dynamic load-balancing strategy is designed into the system. Each client process is started by specifying two parameters on the command line. The first is the name of the computer running the process. This allows the server to provide periodic updates on the status of each client, by name. The second command line parameter is the value "true" or "false". This specifies whether the processor is considered fast (true) or slow (false). Whenever a client processor that has been deemed slow generates 50,000 nodes, in addition to requesting the server to update the best tour value, it queries the server to see whether the server's node pool is empty and whether there exists a fast client processor that is idle (one that has completed the processing of its final node). If the server responds in the affirmative to these two queries, the slow client transfers its priority queue (load) to the server. Each idle fast client pings the server every second for a handoff of a load (priority queue) that may have been transferred to the server from a slow client. The effect is to transfer the remaining load from each slow client to an available fast client with a small delay time because of the requirement that 50,000 nodes has been generated before the slow client hands off its load to the server (a matter of a second or two in worst case). This end-game dynamic load balancing led to much faster overall execution times compared to the simpler (earlier) design that did not include such end-game load balancing.

All of the details of the design outlined above are in the revised class *TSP* (the server class) and revised class *ProcessNodes* (the client class) presented later.

2 LOGISTICS

Setting up an RMI server and clients is a tedious but straight-forward process. The best documentation that this author has found is in the Sun tutorial on RMI that is freely available to all members of the Java Developer Connection (membership is free after registering). A simple HTTP server specifically designed to serve the required classes and stubs needed in the RMI system was also downloaded (free of charge) from Sun's website. A link to this server is provided in the RMI tutorial referenced above. This server is a light-weight http server that allows the user to specify a port and class path from which to serve the needed classes and stubs when the server is launched. This http server was used on each of the client machines as well as the server. As will be evident from the source code provided below, the IP address of the server is needed by each client. In principal, the distributed processing designed in this system could utilize clients available on the internet as long as they could "talk" to the server (without a typical firewall interfering). The IP address could be exchanged for a domain-name URL. This

has not been attempted by the author. The distributed processing was done only on a local area network.

Each client process communicates with the server through a stub class that is compiled on the server using the *rmic* compiler that is part of the standard Java software development kit. This stub class must be available to each client. The details of how this is done is provided in the RMI tutorial. It took this author several hours to master all the details required to get each client to handshake with the server. The server machine must also run an *rmiregistry* server (details again provided in the RMI tutorial). No attempt shall be made in this column to explain these networking details since they would vary from one network setup to another.

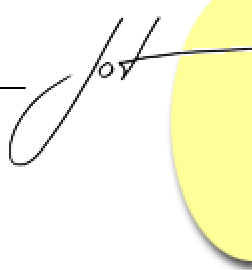
The author's LAN that provides the basis for the distributed process implementation consists of a slow Pentium 3 running Windows NT, a relatively slow Powerbook running Mac OS 10.2.3 and JDK 1.3.1, a fast Pentium 4 running Windows 2000 and JDK 1.4, a fast Pentium 3 running Windows 2000 and JDK 1.4 and a fast Powermac running Mac OS 10.2.3 and a beta version of JDK 1.4. So three operating systems are represented, two Java virtual machines are represented and three processor architectures are represented in this LAN.

3 THE IMPLEMENTATION DETAILS

Class *ProcessNodes* in Listing 1 presents the details of each client process.

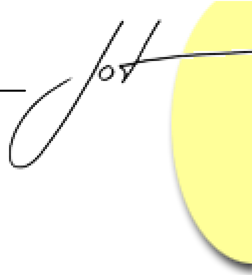
Listing 1 – Client class ProcessNodes

```
import java.util.*;
import java.rmi.*;

public class ProcessNodes {
    // Constants
    static int nodesPerDotAndQuery = 50000;
    static int nodesPerOutputToServer = 1000000;

    // Fields
    private TreeSet queue;
    public int numRows;
    private int numCols;
    private Node bestNode;
    public Cost c;
    private long totalNodeCount = 0L;
    private TSPInterface tsp;
    private int bestTour;
    private String computerName; // User supplied name of computer
    private boolean fastMachine;

    // Commands
    public void remove (Node node) {
```



```
        queue.remove(node);
    }

    public void setQueue (TreeSet queue) {
        this.queue = queue;
    }

    public void processNodes () {
        try {
            bestTour = tsp.bestTour();
            while (queue.size() > 0) {
                Node next = (Node) queue.first();
                if (next.size() == numRows - 1 &&
                        next.lowerBound() < bestTour) {
                    bestTour = next.lowerBound();
                    tsp.output2(computerName, next, totalNodeCount);
                }
                synchronized(queue) {
                    queue.remove(next);
                }
                if (next.lowerBound() < bestTour) {
                    int newLevel = next.level() + 1;
                    byte [] nextCities = next.cities();
                    int size = next.size();

                    for (int city = 2; city <= numRows; city++) {
                        if (!present((byte) city, nextCities)) {
                            byte [] newTour = new byte[size + 2];
                            for (int index = 1; index <= size; index++)
{
                                newTour[index] = nextCities[index];
                            }
                            newTour[size + 1] = (byte) city;
                            Node newNode =
                                new Node(newTour, size +
                                         1, numRows);
                            newNode.setLevel(newLevel);
                            totalNodeCount++;
                            if (totalNodeCount % nodesPerDotAndQuery ==
                                    0) {
                                System.out.print(".");
                                bestTour = tsp.bestTour();
                                if (!fastMachine) {
                                    // Test to see whether there are
                                    // any fast machines idle and
                                    // server pool is empty
                                    if (tsp.poolExpired() &&
                                            tsp.fastMachinesIdle() > 0) {
                                        tsp.transfer(computerName,
                                                     queue);
                                        System.out.println(
                               "Transfering load to a fast machine");
                                        System.exit(0);
                                    }
```

```
                        }
                    }
                    if (totalNodeCount % nodesPerOutputToServer
                            == 0) {
                        tsp.output1(computerName,
                                  totalNodeCount);
                    }
                    newNode.computeLowerBound(c);
                    int lowerBound = newNode.lowerBound();
                    if (lowerBound < bestTour) {
                        synchronized(queue) {
                            queue.add(newNode);
                        }
                    } else {
                        newNode = null;
                    }
                }
            }
        } else {
            next = null;
        }
    }
    if (!tsp.poolExpired()) {
        queue = tsp.getQueue(computerName);
        processNodes();
    } else {
        tsp.registerNodeCount(totalNodeCount);
        tsp.stop(computerName, false);
        try {
            if (fastMachine) {
                tsp.incrementFastMachinesIdle();
                new Thread() {
                    public void run() {
                        try {
                            int count = 0;
                            while (count < 5 &&
                                   !tsp.stopped() &&
                                   tsp.queueIsEmpty()) {
                                sleep(1000);
                                System.out.println("ping");
                                count++;
                            }
                            if (count >= 5) {
                                System.exit(0);
                            }
                            if (!tsp.stopped() &&
                                    !tsp.queueIsEmpty()) {
                              tsp.decrementFastMachinesIdle();
                              queue =
                             tsp.getEntireQueue(computerName);
                             processNodes();
                            }
                        }
```



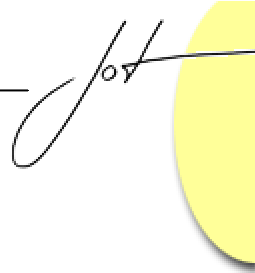
```
                            catch (InterruptedException ex) {
                                System.out.println(ex);
                            }
                            catch (RemoteException e) {
                                System.out.println(e);
                            }
                        }
                    }.start();
                }
            } catch (RemoteException ex) {
                System.out.println(ex);
            }
        }

    } catch (RemoteException ex) {
        System.out.println(ex);
    }
}

public TreeSet queue () {
    return queue;
}

public long totalNodeCount () {
    return totalNodeCount;
}

private boolean present (byte city, byte [] cities) {
    for (int i = 1; i <= cities.length - 1; i++) {
        if (cities[i] == city) {
            return true;
        }
    }
    return false;
}

public static void main (String [] args) {
    if (args.length != 2) {
        System.out.println("Usage: computerName true/false");
        System.exit(1);
    }
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
    }
    try {
        ProcessNodes obj = new ProcessNodes();
        obj.fastMachine = args[1].equals("true");
        obj.computerName = args[0];
        System.out.println(obj.computerName + " starting client
                                computational process.");
        String name = "rmi://www.xxx.yyy.zzz//TSPObject";
        // Replace the www.xxx.yyy.zzz by the actual IP address
        // or domain name of the server
        obj.tsp = (TSPInterface) Naming.lookup(name);
```

```
            obj.queue = obj.tsp.getQueue(obj.computerName);
            obj.numRows = obj.tsp.numRows();
            obj.c = obj.tsp.cost();
            obj.tsp.registerProcess(obj.computerName);
            obj.processNodes();
        } catch (Exception ex) {
            System.out.println(ex);
        }
    }
}
```

Listing 2 contains the needed *TSPInterface* class that contains the signature of all the public methods that each client can invoke on the server (through the stub class).

Listing 2 – Class TSPInterface

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.*;

public interface TSPInterface extends Remote {
    public void stop (String computerName, boolean forced) throws
        RemoteException;
    public int numRows () throws RemoteException;
    public int bestTour () throws RemoteException;
    public Cost cost () throws RemoteException;
    public TreeSet getQueue (String computerName) throws
        RemoteException;
    public int fastMachinesIdle() throws RemoteException;
    public boolean stopped() throws RemoteException;
    public boolean queueIsEmpty() throws RemoteException;
    public boolean poolExpired() throws RemoteException;
    public TreeSet getEntireQueue (String computerName) throws
        RemoteException;
    public void output1 (String computerName, long totalNodeCount)
        throws RemoteException;
    public void output2 (String computerName, Node next, long
                         totalNodeCount) throws RemoteException;
    public void registerProcess (String computerName) throws
        RemoteException;
    public void registerNodeCount (long count) throws RemoteException;
    public void incrementFastMachinesIdle() throws RemoteException;
    public void decrementFastMachinesIdle() throws RemoteException;
    public void transfer (String computerName, TreeSet queue) throws
        RemoteException;
}
```

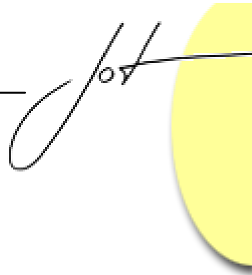


Listing 3 presents the details of the server class *TSP*.

Listing 3 – Server class TSP

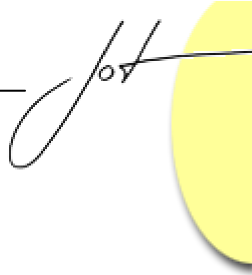
```
/**
  * TSP Branch and Bound Server Class
*/

import java.awt.*;
import java.util.*;
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;

public class TSP extends UnicastRemoteObject
        implements TSPInterface, Serializable {
    // Fields
    private int bestTour = Integer.MAX_VALUE / 4;
    private Node bestNode;
    private TreeSet queue = new TreeSet();
    private int numRows;
    private Cost c;
    private long totalNodeCount = 0L;
    private double elapsedTime = 0.0;
    private TimeInterval t = new TimeInterval();
    private int numberStopped = 0;
    private double accumulatedTime = 0.0;
    private int numberProcesses = 0;
    private int fastMachinesIdle;
    private TSPUI gui;
    private boolean noMore = false;
    private boolean poolExpired = false;

    public TSP (int [][] costMatrix, int size, int bestTour,
                TSPUI gui) throws RemoteException {
        super();
        this.gui = gui;
        this.bestTour = bestTour;
        numRows = size;
        c = new Cost(numRows, numRows);
        for (int row = 1; row <= size; row++) {
            for (int col = 1; col <= size; col++) {
                c.assignCost(costMatrix[row][col], row,
                             col);
            }
        }
    }

    public synchronized void incrementFastMachinesIdle() throws
              RemoteException {
        fastMachinesIdle++;
    }
```

```
public synchronized void decrementFastMachinesIdle() throws
          RemoteException {
    fastMachinesIdle--;
}

public synchronized int fastMachinesIdle() throws RemoteException {
    return fastMachinesIdle;
}

public synchronized void transfer (String computerName, TreeSet q)
        throws RemoteException {
    System.out.println(computerName +
                       " has transferred its load to the server.");
    queue = new TreeSet(q);
}

/* Returns true when the number of processes (clients) to send stop
   equals the total number of processes */
public synchronized boolean stopped () throws RemoteException {
    return noMore;
}


/* Returns true when the queue is emptied the first time */
public synchronized boolean poolExpired () throws RemoteException {
    return poolExpired;
}

/* Returns true when the queue is empty.  It may be empty and then
   re-filed because of transfer from a slow client */
public synchronized boolean queueIsEmpty() throws RemoteException {
    return queue.size() == 0;
}

public synchronized void registerProcess (String computerName)
           throws RemoteException {
    // Start the clock going as soon as the first client kicks in
    if (numberProcesses == 0) {
        t.startTiming();
    }
    System.out.println(computerName + " has started processing.");
    numberProcesses++;
}

public synchronized void output1 (String computerName, long
                                  totalNodeCount)
        throws RemoteException {
    t.endTiming();
    double time = t.getElapsedTime();
    int hours = (int) (time / 3600.0);
    time -= hours * 3600;
    int minutes = (int) (time / 60.0);
```



```
        time -= minutes * 60;
        int seconds = (int) time;
        System.out.println(computerName);
        System.out.println("Elapsed time: " + t.getElapsedTime() +
             " seconds.      <" + hours + " hours " + minutes +
             " minutes " + seconds + " seconds>");
        System.out.println("Nodes generated: " + totalNodeCount);
        System.out.println("Size of server node pool: " +
                           queue.size());
        System.out.println("Best tour cost: " + bestTour);
        System.out.println("Best tour: " + bestNode);
        System.out.println("\n");
    }

    public synchronized void output2 (String computerName, Node next,
                                      long totalNodeCount)
                throws RemoteException {
        if (noMore) {
            return;
        }
        int bestTour = next.lowerBound();
        if (bestTour < this.bestTour) {
            setBestTour(bestTour);
            setBestNode(next);
            t.endTiming();
            double time = t.getElapsedTime();
            int hours = (int) (time / 3600.0);
            time -= hours * 3600;
            int minutes = (int) (time / 60.0);
            time -= minutes * 60;
            int seconds = (int) time;
            System.out.println(computerName);
            System.out.println("Elapsed time: " + t.getElapsedTime() +
                    " seconds.      <" + hours + " hours " + minutes +
                    " minutes " + seconds + " seconds>");
            System.out.println("Nodes generated: " + totalNodeCount);
            System.out.println("Best tour cost: " + bestTour);
            System.out.println("Best tour: " + bestNode);
            if (!poolExpired) {
                System.out.println("Size of server node pool: " +
                                   queue.size());
            } else {
                System.out.println("Size of server node pool: " + 0);
            }
            System.out.println();
        }
    }

    public synchronized TreeSet getQueue (String computerName)
                throws RemoteException {
        if (noMore) {
            return null;
        }
        System.out.println(computerName +
```

```
        " attempting to get node from the server's node pool of size "
        + queue.size() + ".");
        if (queue.size() == 0) {
            return null;
        }
        TreeSet t = new TreeSet();
        Node n = (Node) queue.first();
        t.add(n);
        synchronized(queue) {
            queue.remove(n);
        }
        poolExpired = queue.size() == 0;
        return t;
    }

    public synchronized TreeSet getEntireQueue (String computerName)
                throws RemoteException {
        System.out.println("Load transfered from server to " +
                           computerName);
        TreeSet q = new TreeSet(queue);
        queue.clear();
        return q;
    }

    public synchronized void stop (String computerName, boolean forced)
                throws RemoteException {
        if (forced && noMore) {
            return;
        }
        if (!forced) {
            System.out.println(computerName + " completed its work.");
        }
        numberStopped++;
        if (numberStopped == numberProcesses || forced) {
            t.endTiming();
            noMore = true;
            if (!forced) {
                System.out.println("\n\nOptimum solution obtained.");
            } else {
                System.out.println(
      "Solution forced to stop prematurely and may not be optimum.");
            }
            System.out.println(
                    "The total number of nodes generated: " +
                    totalNodeCount);
            System.out.println("Tour cost: " + bestTour);
            double time = t.getElapsedTime() + accumulatedTime;
            int hours = (int) (time / 3600.0);
            time -= hours * 3600;
            int minutes = (int) (time / 60.0);
            time -= minutes * 60;
            int seconds = (int) time;
            System.out.println("Elapsed time: " + (t.getElapsedTime() +
```

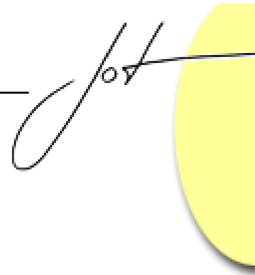


```
                    accumulatedTime) + " seconds.      <" + hours +
                    " hours " + minutes + " minutes " + seconds +
                    " seconds>");
            try {
                gui.displayOutput();
            } catch (Exception ex) {}
        }
    }

    public void setBestTour (int bestTour) {
        if (bestTour < this.bestTour) {
            this.bestTour = bestTour;
        }
    }

    public void setBestNode (Node bestNode) {
        this.bestNode = bestNode;
    }

    public void registerNodeCount (long count) throws RemoteException {
        totalNodeCount += count;
    }

    public void generateSolution (boolean ongoing) {
        if (!ongoing) {
            // Create root node
            byte [] cities = new byte[2];
            cities[1] = 1;
            Node root = new Node(cities, 1, numRows);
            root.setLevel(1);
            totalNodeCount++;
            root.computeLowerBound(c);
            System.out.println(
                   "The lower bound for root node (no constraints): " +
                    root.lowerBound());
            queue.add(root);
            Node next = (Node) queue.first();
            synchronized(queue) {
                queue.remove(next);
            }
            int newLevel = next.level() + 1;
            byte [] nextCities = next.cities();
            int size = next.size();

            for (int city = 2; city <= numRows; city++) {
                if (!present((byte) city, nextCities)) {
                    byte [] newTour = new byte[size + 2];
                    for (int index = 1; index <= size; index++) {
                        newTour[index] = nextCities[index];
                    }
                    newTour[size + 1] = (byte) city;
                    Node newNode =
                        new Node(newTour, size + 1, numRows);
                    newNode.setLevel(newLevel);
```

```
                        totalNodeCount++;
                        newNode.computeLowerBound(c);
                        int lowerBound = newNode.lowerBound();
                        queue.add(newNode);
                    }
                }
            }
        }

    public Node bestNode () {
        return bestNode;
    }

    public int bestTour () {
        return bestTour;
    }

    public int numRows () throws RemoteException {
        return numRows;
    }

    public long nodesGenerated () {
        return totalNodeCount;
    }

    private boolean present (byte city, byte [] cities) {
        for (int i = 1; i <= cities.length - 1; i++) {
            if (cities[i] == city) {
                return true;
            }
        }
        return false;
    }

    public Cost cost () {
        return c;
    }
}
```

The GUI class (details not shown) contains the code that binds the TSPObject in the rmi registry. This is the object that is used in each client class for inter-process communication between the client and server. The code segment that accomplishes this is (replace the IP address www.xxx.yyy.zzz with the actual IP address of the server)

```
solution = new TSP(costMatrix, size, Integer.MAX_VALUE / 4, this);
try {
   Naming.rebind("rmi://www.xxx.yyy.zzz//TSPObject", solution);
   System.out.println(
       "RMI server running with TSP object bound in rmi registry");
} catch (MalformedURLException ex) {
    System.out.println(ex);
}
```

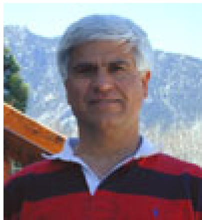


4 EMPIRICAL RESULTS

The computation time was reduced as one would expect by the presence of additional processors. The result below shows the benefit of the distributed processing in solving the 24 city problem.

Number of Cities	**Execution Time** on Dell 1.7GHz (Single Threaded Previous Implementation)	**Execution Time** Using All Five Machines in parallel
24	**299.25** seconds	**73.415** seconds

About the author



Richard Wiener is Associate Professor of Computer Science at the University of Colorado at Colorado Springs. He is also the Editor-in-Chief of JOT and former Editor-in-Chief of the Journal of Object Oriented Programming. In addition to University work, Dr. Wiener has authored or co-authored 21 books and works actively as a consultant and software contractor whenever the possibility arises.