JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 3, March-April 2004

Asynchronous RMI for CentiJ

Douglas Lyon, Computer Engineering Department, Fairfield University, Connecticut, USA

Abstract

CentiJ is a software synthesis system that, until recently, used synchronous, semi-automatic static proxy delegation to help in the automation of the creation of distributed Java programs on NOWS (Networks of Workstations). This paper reports our recent extension to CentiJ so that invocations are asynchronous. Further, we have achieved transparency with respect to local vs. non-local asynchronous invocations so that software can be properly tested in a local mode.

Reflection helps in the creation of bridge pattern code (i.e., interfaces and proxies) for asynchronous message forwarding via RMI.

The CentiJ technique improves programmer productivity by automating the creation of the housekeeping code. The use of compile-time static delegation enables type-safety. CentiJ leaves the part of the code that forms the core computation unchanged. It generates new code that enables asynchronous invocations via the observer-observable design pattern.

1 INTRODUCTION

RMI supports an object-oriented communication framework for distributed computation in a heterogeneous network on a remote address space [1][2]. It can be used between systems written in different languages [3]. Experience has shown that the use of RMI can require significant programmer effort and the writing of extra source code. The goal of the *CentiJ* project is to make RMI easier to use.

CentiJ enables the remote invocation of existing, (i.e., *legacy*) code. It wrappers the communications through a remotely invoked *bridge*. Communications are encapsulated in the bridge code so that the programmer does not need to modify the legacy code, opting, instead, for automatic generation of bridge code. The bridge code reuses original implementations and provides a means of computation distribution.

The RMI Problem

The RMI problem can be broken down into three sub-problems. The first is called the *legacy bridge problem*. The second is called the *virtual proxy synthesizer* problem. The third problem, which is the focus of this paper, is the *asynchronous invocation problem.*

The legacy bridge problem may be stated as follows: given a large number of methods in a variety of classes, build a bridge to these methods so that there is a reuse of the implementations in the existing code. We are subject to the constraint that we cannot change the existing code. Further we may not even have the existing source code. The legacy bridge problem is solved by building code that implements the *bridge pattern*. The *bridge code* consists of an interface, or protocol of communication and an implementation of the communication. Legacy code is often fragile, hard to maintain, difficult to reverse engineer, unchangeable and sometimes poorly designed. Hence the constraint that we cannot change the legacy code. Subject to these constraints, *CentiJ* builds a bridge between new code and the legacy system. Thus providing a solution to the *legacy bridge problem.*

The *virtual proxy problem* is the second sub-problem solved by *CentiJ*. With the virtual proxy, the goal is to method-forward to an existing implementation. *CentiJ* uses inputs from the legacy bridge problem and generates code that can be invoked on a remote address space.

The *asynchronous invocation problem* can be stated as follows: given a set of synchronous invocations, with synchronous returns, find a design pattern that enables asynchronous invocations with asynchronous returns. The alternative is to block the invokers' thread of execution.

CentiJ addresses the above problems, by creating an observable virtual proxy. Multi-threaded invocations to the remote code are executed on the master host. Callback is performed for each invocation so that returns can be supplied to the observers.

What is wrong with RMI?

RMI code is often written manually. This requires an extensive analysis of the existing code. Typically, a large number of dependencies between classes complicates the analysis [4].

To illustrate the problems with RMI, consider the following eight-step procedure for manually writing an RMI program:

1. Define an interface(s) for the remote class(s). Compile the interface.
2. Create and compile classes that implement the interfaces.
3. Use the Java rmic compiler to create stub class(es) from the implementation class(es).
4. Create a server application and compile it
5. Start the *rmiregistry* .
6. Start the server application.
7. Create a client program that accesses the remote interface(s). Compile the client program.
8. Test the client program.

These 8 steps are critical to the successful completion of the RMI system. Failure for any single step will result in run-time exceptions being thrown. Additionally, the code is encumbered with RMI artifacts that complicate maintenance and make the code less readable. Other problems will become evident as the example unfolds:

Step 1. Define an interface(s) for the remote class(s). Compile this interface

```
public interface RemoteHello extends Remote {
  public String getMsg() throws RemoteException;
}
```

Aside from the fact that all the code is created manually, the API of the *getMsg* method now must throw a *RemoteException*. This is probably not something the original code had to do.

Step 2. Create and compile classes that implement the interfaces.

```
public class RemoteHelloImplementation
        extends UnicastRemoteObject
 implements RemoteHello {
    private String msg
        = "Hello world";
    public RemoteHelloImplementation()
            throws
    RemoteException {
    }
   public String getMsg()
      throws RemoteException {
       return msg;
   }
}
```

Some problems here include all those mentioned in step 1 and a problem unique to object-oriented languages that lack multiple inheritance (like Java); by subclassing the *UnicastRemoteObject* we are no longer able to subclass anything else. In fact, due to the decreased reliability of RMI, it is probably a good idea to test implementations using non-remote classes.

Step 3. Use the Java rmic compiler to create stub class(es) from the implementation class(es). In this step the programmer changes directories to the location of the class files (unless the class path has been set) then types:

```
rmic -v1.2 -d . net.rmi.simpleExample.RemoteHelloImplementation
```

This is bad news for several reasons. First the programmer has to *remember* to do something! This is really HARD. Programmers should never be called upon to insert actions into the programmer cycle (edit, compile, test). Also, if the programmer forgets, the code will run with old stubs and this can lead to cyptic run-time errors *occasionally*. This is the worst kind of unreliability. Finally, if the programmer does not have a correct classpath, or forgets to change to the correct location for the class files, then the *rmic* invocation will fail.

Step 4. Create a server application and compile it. The programmer hand codes the following program:

```
public class RmiHelloServer {
// before you run this program,
// you must start the rmiregistry
// from the classpath root.
    public static void
       main(String args[]){
        try {
            startServer();
   }
  catch (RemoteException e) {
            e.printStackTrace();
   }
  catch (MalformedURLException e) {
            e.printStackTrace();
   }
}
private static void startServer()
            throws
     RemoteException,
     MalformedURLException {
        println("starting server");
    RemoteHello ro = new
     RemoteHelloImplementation();
  println("binding remote instances");
  Naming.rebind("RemoteHello",ro);
  println("waiting for invocations");
}
public static void
   println(Object o){
        System.out.println(o);
   }
}
```

The first problem we notice is that a string “RemoteHello” must be correct and bound to the *RemoteHelloImplementation*. If this string were typed improperly it results in a run-time exception being thrown. A different run-time exception is thrown if the programmer attempts to run *RemoteHello* without running the *rmiregistry* first.

Step 5. Start the *rmiregistry* . In this step, the programmer either sets the class path or changes to the location of the pre-compiled classes. Then types:

```
rmiregistry
```

This step has the same problems as step 3.

Step 6. Start the server application. The programmer types:

```
java RmiHelloServer
```

This step also has the same problems as step 3.

Step 7. Create a client program that accesses the remote interface(s). Compile the client program.

```
public class RmiHelloClient {
    private String rmiUrl = null;
    private RemoteHello rh = null;
public RmiHelloClient(
       String location) {
       rmiUrl = location;
 try {
        rh = lookupDelgate();
 } catch (NotBoundException e) {
        e.printStackTrace();
 } catch (MalformedURLException e) {
        e.printStackTrace();
 } catch (RemoteException e) {
       e.printStackTrace();
 }
}
private RemoteHello lookupDelgate()
    throws NotBoundException,
            MalformedURLException,
            RemoteException {
 RemoteHello rh = (RemoteHello)
    Naming.lookup(rmiUrl);
 return rh;
}
public static void
     main(String args[]) {
 try {
  RmiHelloClient rhc =
       new RmiHelloClient(
  "rmi://localhost/RemoteHello");
      rhc.testGetMsg();
 }
 catch (RemoteException e) {
   e.printStackTrace();
 }
}
private void testGetMsg()
       throws
      RemoteException {
 System.out.println(rh.getMsg());
 }
}
```

The *sayHello* method, as implemented above, has several problems. First, the remote location of the server is hard-coded into the main as *"rmi://localhost/"*. That is a parameter that will have to change, once the location of the remote host is known. Secondly, there is a lot of housekeeping in the *RmiHelloClient.* During construction, it must bind the *RemoteHello* interface to the remote implementation using a lookup feature. Also, all remote invocations can throw *RemoteExceptions* at run-time (something the non-remote code never had to do).

Step 8. Test the client program.

```
java RmiHelloClient
```

This last step can fail and emit run-time errors that may be unclear. It also is not the end of the story. Now suppose that you seek to run the program on a different machine. This brings us to the deployment issues, which are both difficult and beyond the scope of this paper.

As we observe the creation of the above RMI code, we can characterize the invocations as being synchronous invocations with synchronous returns. That is, any invocation to any method will block the callers' thread of execution.

2 VARIOUS BRIDGE IMPLEMENTATIONS

This section examines the various implementations of the bridge pattern. The alternatives are based in *delegation*. We describe the two types of delegation, dynamic and static. Dynamic delegation is delegation that is performed at run-time using dynamic class loading. Static delegation is delegation that is performed at compile time. *CentiJ's* static delegation technique generates Java source code that must be compiled to be used.

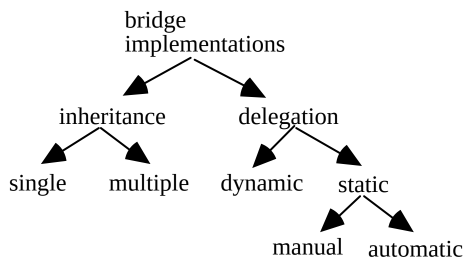

Figure 1. Various bridge implementations

Figure 1 shows the various bridge implementations. While manual static delegation is the most common, type-safe implementation of a bridge, it is also the most labor intensive. The new mode of *automatic static delegation* alters the economics of static delegation so that it is both type-safe and low-cost.

3 THE CENTIJ SOLUTION

CentiJ addresses some of RMI's problems by using reflection to automate several of the steps described in section 1. Aside from the house keeping (like running RMIC, generating interfaces and RMI wrappers) the core idea behind CeniJ is that it uses delegation. For example, rather than modify an existing *HelloWorld* program, we pass an instance of the core-computation class to *CentiJ's* generator. For example:

```
public class HelloWorld {
    public HelloWorld() {
    }
public static void
    main(String args[]) {
        HelloWorld hw =
                new HelloWorld();
        hw.testGetMsg();
    }
public String getMsg() {
        return "Hello world";
    }
public void testGetMsg() {
        System.out.println(getMsg());
 }
}
```

HelloWorld is simple, well-tested, well-understood, locally-invoked code. It is generally better software engineering to start with a working local program and keep it, unmodified, for the purpose of testing. CentiJ automatically generates the needed interface, along with the stubs. The following shows automatically generated code where only the public, dynamic methods are message forwarded to the existing *HelloWorld* class:

```
public class RemoteHelloImplementation
        extends UnicastRemoteObject
        implements RemoteHello {
 HelloWorld hw = new HelloWorld();
public RemoteHelloImplementation()
        throws RemoteException {
}
public void testGetMsg() throws RemoteException {
   hw.testGetMsg();
}
public String getMsg() throws RemoteException {
        return hw.getMsg();
 }
}
```

There is disagreement about what delegation is (and is not). According to one definition, delegation uses a receiving instance that forwards messages (or invocations) to its delegate(s). This is sometimes called a *consultation* [5]. This is the definition that we use in *CentiJ*.Variations on delegation give rise to several *design patterns*. For example, if methods are forwarded without change to the interface, then you have an example of the *proxy pattern*. If you simplify the interface with a subset of methods to a set of delegates, then you have a *facade pattern*. If you compensate for changes (i.e., deprecations) in the delegates, and keep the client classes seeing the same contract, then you have the *adapter* *pattern*. Thus, we define *static delegation* as compile-time, type-safe, message forwarding from a proxy class to some delegate(s).

Finally, on the client-side, a means for asynchronously invoking methods is needed. Because we are adding a new responsibility to the proxy class we are making use of the *decorator pattern* [6]. The new responsibility of the proxy class is to keep track of those instances that are interested in the method's results. Thus the generated proxy class makes use of the *observer-observable* design pattern. In order to illustrate the observer-observable design pattern, we present the following asynchronous version of the *HelloWorld* class:

```
public class ASynHelloWorld extends Observable {
    HelloWorld hw = new HelloWorld();
public ASynHelloWorld() {
}
public void getMsg() {
  Thread t
   = new Thread(new Runnable() {
  public void run() {
   Object o = hw.getMsg();
   setChanged();
   notifyObservers(o);
   }});
 t.start();
}
public void testGetMsg() {
  Thread t
    = new Thread(new Runnable() {
        public void run() {
   hw.testGetMsg();
    }});
   t.start();
 }
}
```

All returns are communicated via the *update* method, as defined in the *Observer* interface. An example *Observer* follows:

```
public class ASynHelloWorldTest
        implements Observer {
public ASynHelloWorldTest() {
}
public void update(
   Observable obs,
   Object arg){
  System.out.println(arg);
}
public static void
  main(String args[]){
 ASynHelloWorld ashw
    = new ASynHelloWorld();
    ASynHelloWorldTest ashwt
       = new ASynHelloWorldTest();
          ashw.addObserver(ashwt);
          ashw.getMsg();
          ashw.testGetMsg();
    }
  }
```

The *main* method is used to instance the classes and hook up the observer with the observable. Thus it is an example of the *mediator* design pattern [6]. Note that there is no attempt at transparency in calling the remote code. The core remote implementations have been left untouched, but invoking them asynchronously requires that we alter the means by which the methods' are invoked. The alternative is to block the invoking methods thread of execution. The following is an example of the asynchronous RMI version of the asynchronous *HelloWorld* class. Note that the interface is identical:

```
  public class ArmiHelloWorld
     extends Observable {
     RmiHelloClient rmiHwClient
             = new RmiHelloClient(
  "rmi://localhost/RemoteHello");
  public ArmiHelloWorld() {}
  public void getMsg() {
      Thread t =
      new Thread(new Runnable() {
       public void run() {
        Object o = null;
        try {
          o = rmiHwClient.getMsg();
        } catch (RemoteException e) {
          e.printStackTrace();
       }
       setChanged();
       notifyObservers(o);
      }});
     t.start();
   }
  public void testGetMsg() {
      Thread t = new Thread(new Runnable() {
           public void run() {
   try {
    rmiHwClient.testGetMsg();
      } catch (RemoteException e) {
      e.printStackTrace();
      }
    }});
    t.start();
   }
  }
```

Thus, to modify the client to take advantage of the remote version of this class means that only the class name be changed:

```
public class ArmiHelloWorldTest
        implements Observer {
public ArmiHelloWorldTest() {}
public void update(
    Observable obs,
    Object arg){
System.out.println(arg);
 }
public static void
    main(String args[]){
  ArmiHelloWorld ashw
      = new ArmiHelloWorld();
  ArmiHelloWorldTest ashwt
      = new ArmiHelloWorldTest();
  ashw.addObserver(ashwt);
  ashw.getMsg();
  ashw.testGetMsg();
 }
}
```

The locally invoked, asynchronous classes, have an identical API to the remotely invoked, asynchronous classes. Thus we have a kind of transparency between the local and remote versions of the code. This is known as the Liskov principle [7].

4 RELATED WORK

There are several projects that aim at making Java programs parallel. One example is the *Do!* project [8]. The *Do!* project does not use a static refactoring of the code to help with distributions, instead it uses special kinds of distributed *collections* to explicitly express concurrency.

Another tool, *Orca*, automated distribution decisions using a run-time system for placement and replication selection for remote jobs [9]. The *Ninja* project uses clusters of workstations, active proxies and low-level byte code specialization for fine-grained parallelism. The *Pangaea* system uses a static source code analysis and a middleware back-end to distribute centralized Java programs. *J-Orchestra* takes the approach of fine-grained automatic parallelism using byte-code output from the Java compiler. *J-Orchestra*, *Do!*, *Orca*, *Ninja* and *Pangaea* do not attempt to perform any type of refactoring or code generation. Also, they try to automate the decision for placing programs on other systems (a decision that is hard to automate). Their fine-grained approach to automating parallelism does not take into account the programmers' input (which often stems from specialized knowledge about the problem domain and code structure) [10][11][12][13].

RMI automation is not new. *JavaParty* has been around for some time [14]. However, it requires that the language be modified. Further, it does not gather instances to build bridges as *CentiJ* does.

Fanta and Rajlich have also worked on altering existing code, by moving functions around, expelling them from classes, refactoring properties and updating invocations to these elements. Moore has also worked on automatic refactoring and method restructuring. This work refactors expressions from methods. The Guru tool of Moore automatically refactors common code out of methods into abstract super-classes. For programming languages that lacks multiple inheritance (like Java) this effort can adversely affect how methods can be shared [15]. Casais claims that there may not be any case studies on the automatic reorganization of class hierarchies [16]. Thus, the question of how the code quality is changed by these systems remains open.

The *CentiJ* approach to automating the synthesis of bridge code is like the pre-processor approach of the *Jamie* system [17]. A problem with *Jamie* is that it extends the language by creating a macro-preprocessor. Also, *Jamie* uses *dynamic* delegation.

The LAVA language extends Java to provide for delegation. Kniesel says that current implementations of LAVA have an efficiency that is unacceptable [18][19]. In comparison, *CentiJ*'s static delegation is subject to in-line expension, at compile time. This should generally be faster than dynamic proxies, as it is a pay-now or pay-later approach. The static compilation costs that CentiJ incurs are paid up-front. In theory, therefore, with in-lining enabled, there should be *no performance degradation* (though this remains to be proven).

Fisher and Mitchell provide a new delegation-based language [20]. The primary advantage of the Fisher-Mitchell system is its ability to infer type, and it's ability to resolve method names at compile-time. They had to devise a new language for this. In comparison, *CentiJ* works by API extension, rather than by creating a new language. An API extension is easier to deploy into an existing environment than a new language.

Delegation has been cited as a mechanism to obtain implementation inheritance via composition [21] [22]. Lieberman introduced delegation in a prototype-based object model in 1986 [23]. He indicated that delegation is considered safer than inheritance because it forces the programmer to select which method to use when identical methods are available in two delegate classes. Systems, like *Kiev*, extend the Java language so that it has multiple inheritance of implementation http://www.forestro.com/kiev/kiev.html. Such language extensions are non-standard and not portable.

Reverse engineering programs, such as *Lackwit*, are able to discover inheritance relationships with greater ease than composition associations [24]. That is because the inheritance association implies a specialization semantic. On the other hand, composition association scales better than single inheritance.

Message forwarding is an implementation sharing mechanism [25]. Experts have disagreed on this point, saying that delegation is a form of class-inheritance (since the execution context must be passed to the delegate). I take the opposite view, as class-inheritance type of sharing of context involves name sharing, property sharing and method sharing. Sharing via delegation is instance sharing. The semantics of instance sharing enable a control of the coupling between instances. This provides a mechanism for reuse without introducing uncontrolled cohesion (which increases brittleness in the code) [26].

Tim Lavers published a technique for automatically generating RMI source code [27]. It is very close to what *CentiJ* presents except that it does not gather the instances to build a bridge class, and makes use of dynamic proxy invocation. Also, it does not support asynchronous invocations.

In summary, all the refactoring systems reviewed in this section (except [26]) not only need to read the source code, but they are like the *Elbereth* system in that they alter source code [4]. In the literature that we have reviewed, we have yet to find a means for automatically creating the bridges created by *CentiJ*. A macro system (or templates) would be a logical means of providing this ability, but this would require a modification of Java.

Methods for automatically generating adapters are not new. In fact, C++ has had a template feature for years [28]. Java has a template feature, called *generics* as part of the draft release of JDK 1.5. The question of which is better, adding some API calls to generate source code, or using generics, remains open.

Asynchronous RMI is not new. Rajeev et al. explored it in their *ARMI* mechanism. Their approach is different from *CentiJ's* in that they do their callbacks directly from the server. In comparison, *CentiJ's* callbacks are local. Further, *ARMI* uses the *Future* class that inserts a return value when it becomes available. It is up to the client to poll the *Future* instance to determine when a result is present, thus the system is not based in notifications, like *CentiJ's* [29].

The *Reflective Remote Method Invocation* (RRMI) system of Thiruvathukal et al. is very close to the *CentiJ* approach. Like *CentiJ* it makes use of reflection and provides a mechanism for asynchronous invocation. Unlike *CentiJ*, RRMI uses dynamic proxies, requiring run-time reflection to do the remote invocation. Worse, still, the strings that describe the method names become embedded in the invoking program. This appears to be both manual and error prone. It also not type safe. Finally, their code examples contain many RMI artifacts, are surrounded by try-catch blocks. On the other hand, they do use notifications, like *CentiJ* does, in order to obtain their results [30].

The *DeJay* system calls remote objects in a asynchronous manner, like *CentiJ* does. However, it relies upon a polling mechanism to determine when a result is ready. Additionally, DeJay uses a compiler for its code pre-processor. The question of which is better, the use of a special compiler (dejayc) or an API extension (like CentiJ does) remains open [31].

5 CONCLUSIONS

CentiJ does asynchronous method forwarding across a transport layer using automatically generated code. It provides locally invocable versions of the asynchronous code, using the observer-observable design pattern. The local version of the code has an API that is close to the remote version of the code, and this helps with local testing, before deployment. It also helps to isolate client code from RMI artifacts.

CentiJ uses delegation with a static binding. This enables inlining of code. Thus static delegation does not suffer from the performance degradation of dynamic delegation.

In brief:

1. Dynamic delegation is more automatic than static delegation.
2. Dynamic delegation is not type-safe, but static delegation is.
3. Automatic static delegation is almost as automatic as dynamic delegation, and just as type safe as static delegation.

The choice between static and dynamic delegation is a choice between safety and flexibility. [32]. Thus, *CentiJ* is a proxy generator that can work without source code from the core computation (using reflection). *CentiJ* is an automatic system, and this can lead to a more reliable means of deployment.

The question of how to select a class for remote invocation remains open.

The question of how the code quality is changed by *CentiJ* remains open.

CentiJ uses a *contract network protocol*. Such a protocol defines a static interface that could help keep API deprecations from propagating to existing code. The question of how well this will work in the face of API deprecations is a topic of future research.

Distributed computation on an unreliable network is an open problem. Also open is the problem of how to best dynamically load balance a computation. It is well known that screen saver-type volunteer computations can be successful; however writing portable screen savers in Java is not easy. This is true, in part, because the screen saver must detect when the machine is not in use (a platform specific activity, a present).

REFERENCES

[1] Doug Lea, *Concurrent Programming in Java, Design Principles and Patterns*, Addison Wesley, Reading, MA. 1997.

[2] Jennings, N., and Wooldridge, M. (2000) “Agent-Oriented Software Engineering”. In *Handbook of Agent Technology* (ed. J. Bradshaw) AAAI/MIT Press. (to appear) http://citeseer.nj.nec.com/wooldridge99agentoriented.html

[3] Aleksander Slominski, M. Govindaraju, D. Gannon and R. Bramley, “SoapRMI C++/Java 1.1: Design and Implementation”, pre-print http://citeseer.nj.nec.com/467360.html

[4] W. Korman and W. G. Griswold. “Elbereth: Tool support for refactoring Java programs”. *Technical report*, University of California, San Diego Department of Computer Science and Engineering, May 1998. http://citeseer.nj.nec.com/korman98elbereth.html

[5] Günter Kniesel: “Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems”. *Technical report IAI-TR-94-3*, Oct. 1994, University of Bonn, Germany. http://citeseer.nj.nec.com/kniesel95implementation.html

[6] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns*, Addison-Wesley, Reading, MA. 1995.

[7] A. C. Myers, J. A. Bank, and B. Liskov: “Parameterized Types in Java”, In: *Proc. of 24th POPL*, pages 132-145, 1997

[8] P. Launay, J.-L. Pazat. “A framework for parallel programming in Java”. In HPCN'98, LNCS, April 1998 http://citeseer.nj.nec.com/launay97framework.html

[9] Bal, H.E., et al. “Performance Evaluation of the Orca Shared-Object Systems” *ACM Trans. CS*, Vol. 16, No. 1 (1998) pages. 1-40.

[10] Eli Tilevich, “J-Orchestra: Automatic Java Application Partitioning”, pre-publication http://citeseer.nj.nec.com/473381.html

[11] Andre Spiegel. Pangaea: “An automatic distribution front-end for Java”. In *Fourth IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments* (HIPS '99), in Proc. IPPS/SPDP '99, San Juan, Puerto Rico, USA, April 1999. IEEE. http://citeseer.nj.nec.com/spiegel99pangaea.html

[12] Andre Spiegel, “Automatic Distribution in Pangaea”, CBS 2000, Berlin, April 2000. See also http://www.inf.fu-berlin.de/~spiegel/pangaea/

[13] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Josheph, R. H. Katz, Z. M. Mao, S. Ross, and B. Zhao. “The Ninja Architecture for Robust Internet-Scale Systems and Services”. *Special Issue of Computer Networks on Pervasive Computing*, 2000. (to appear). http://citeseer.nj.nec.com/gribble00ninja.html

[14] Michael Philippsen and Matthias Zenger. “JavaParty - transparent remote objects in Java”. *Concurrency: Practice and Experience*, 9(11):1125--1242, November 1997. see http://citeseer.nj.nec.com/philippsen97javaparty.html, http://wwwipd.ira.uka.de/JavaParty/tour.html

[15] I. Moore. “Automatic inheritance hierarchy restructuring and method refactoring”. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 235--250, October 1996. SIGPLAN Notices, 31(10). http://citeseer.nj.nec.com/moore96automatic.html

[16] E. Casais, "Automatic reorganization of object-oriented hierarchies: a case study", Object Oriented Systems, 1 (1994), pp. 95-115

[17] John Viega and Bill Tutt and Reimer Behrends, "Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages", CS-98-03, Microsoft Corporation, Feb., 1998, http://citeseer.nj.nec.com/3325.html

[18] Günter Kniesel: "Delegation for Java: API or Language Extension?". Technical report IAI-TR-98-5, May, 1998, University of Bonn, Germany. http://citeseer.nj.nec.com/kniesel97delegation.html

[19] Günter Kniesel, "Type-Safe Delegation for Run-Time Component Adaptation", in R. Guerraoui (Ed.): *Proceedings of ECOOP99.* Springer LNCS 1628. http://citeseer.nj.nec.com/kniesel99typesafe.html

[20] K. Fisher and J. C. Mitchell. "A Delegation-based Object Calculus with Subtyping", in *Proc. of FCT, volume 965 of Lecture Notes in Computer Science*, pages 42-61. Springer-Verlag, 1995. http://citeseer.nj.nec.com/104746.html

[21] Henry Leiberman. "Using prototypical objects to implement share behavious in object-oriented systmes", in *Object-oriented Programming Systems, languages and Applications Conference Proceedings*, pages 214-223.

[22] Johnson and Zweig. "Delegation in C++", in *Journal of Object-Oriented Programming*, 4(11): 22-35, November 1991.

[23] Henry Leiberman. "Using prototypical objects to implement share behavious in object-oriented systms", in *Object-oriented Programming Systems, languages and Applications Conference Proceedings*, pages 214-223.

[24] O'Callahan,R., and Jackson, D., "Lackwit: A program understandingtool based on type inference", in *Proceedings of the 1997 International Conference on Software Engineering* (ICSE'96) (Boston, MA, May 1997), pages 338-348. http://citeseer.nj.nec.com/329620.html

[25] Günter Kniesel, private e-mail communications, kniesel@cs.uni-bonn.de, 2001

[26] D. Bardou and C. Dony. "Split Objects: A Disciplined Use of Delegation Within Objects", in *Proceedings of OOPSLA'96*, San Jose, California. Special Issue of ACM SIGPLAN Notices (31)10, pages 122-137, 1996. http://citeseer.nj.nec.com/bardou96split.html

[27] Tim Lavers, "Java Tip 108: Apply RMI autogeneration", http://www.javaworld.com/javaworld/javatips/jw-javatip108.html

[28] Bjarne Stroustrup. *The C++ programming Language*, Addison-Wesley, Reading, MA. 1991.

[29] Rajeev R. Raje, Hoseph I. William, and Michael Boyles. "An Asynchronous Remote Method Invocation (ARMI) Mechanism for Java". *ACM* http://citeseer.nj.nec.com/467569.html

[30] George K. Thiruvathukal, Lovely S. Thomas, and Andy T. Korczynski. “Reflective remote method invocation”. *Concurrency: Practice and Experience*, 10(11-13):911-926, September-November 1998. http://citeseer.nj.nec.com/thiruvathukal98reflective.html

[31] Marko Boger, Frank Wienberg, and W. Lamersdorf. Dejay: “Unifying concurrency and distribution to achieve a distributed Java”, in *Proceedings of TOOLS Europe '99*, Nancy, France, June 1999. Prentice Hall. http://citeseer.nj.nec.com/boger99dejay.html. See http://www.dejay.org for more details.

[32] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. “Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance”. In *ECOOP '93 Conference Proceedings*, pages 247-267. Kaiserslautern, Germany, July 19.

About the author

After receiving his Ph.D. from Rensselaer Polytechnic Institute, **Dr. Lyon** worked at AT&T Bell Laboratories. He has also worked for the Jet Propulsion Laboratory at the California Institute of Technology. He is currently the Chairman of the Computer Engineering Department at Fairfield University, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. E-mail Dr. Lyon at Lyon@DocJava.com. His website is http://www.DocJava.com.