NOPL White Paper

Daniel Faltyn dtf2110@cs.columbia.edu COMS W4115 Spring 06

Introduction

NOPL is a **N**etwork **O**riented **P**rogramming **L**anguage that makes functions available over a network[1]while abstracting the entire network stack from the developer. Server-side programmers write functions using basic language constructs such as primitives, operators, and control structures.[2] The language inherently makes the functions available over a network.

The NOPL compiler generates a .java file that is a socket-based implementation of the function. When compiled and run in Java, the program listens on a specified port and executes the defined function when it receives a socket request. A side-effect of the compilation is the generation of a .java proxy that clients can use to invoke the remote method(s.)

NOPL is designed for distributed applications that require data in a point-to-point fashion;[3] i.e. clients call functions whenever they need to retrieve or manipulate data. For example, an address book java program may want to search for a person given a first and last name. A developer can write such a function in NOPL, compile and run the generated server as a Java executable, and give the generated proxy to the interested client(s.) The client can then call the function as if it were available in the local process space.

Background

The designer wanted to develop a language that various groups at an investment bank could use to build distributed systems. At Goldman Sachs, the Product Data Quality Group (PDQ) maintains the assets traders buy and sell, trader-support maintains a list of trader positions, HR maintains information about the traders, and Compliance oversees all the groups.

Java client applications in each of the areas need to query some or all of the services to get detailed information or perform a calculations. Figure 1 shows conceptually what developers would do to accommodate the given groups.

Developers first write three servers in NOPL: Trade Details Service, Trader Details Service and Product Details Service. The NOPL compiler generates three java files

[1] A network is any series of connected computers communicating via sockets. E.g. The Internet or a single machine.
[2] If time permits in the project, the designer will include database connectivity and file system manipulations. See the Language Reference Manual for detailed information on all functions.
[3] As opposed to a publish-subscribe paradigm where subscribed clients get data whether they request it or not.

labeled "Separate Servers" that when further compiled and run in java, are immediately available on the network.

Client developers can embed any of the needed proxies (indicated with the name '%Proxy' in the diagram) into their proprietary client code.

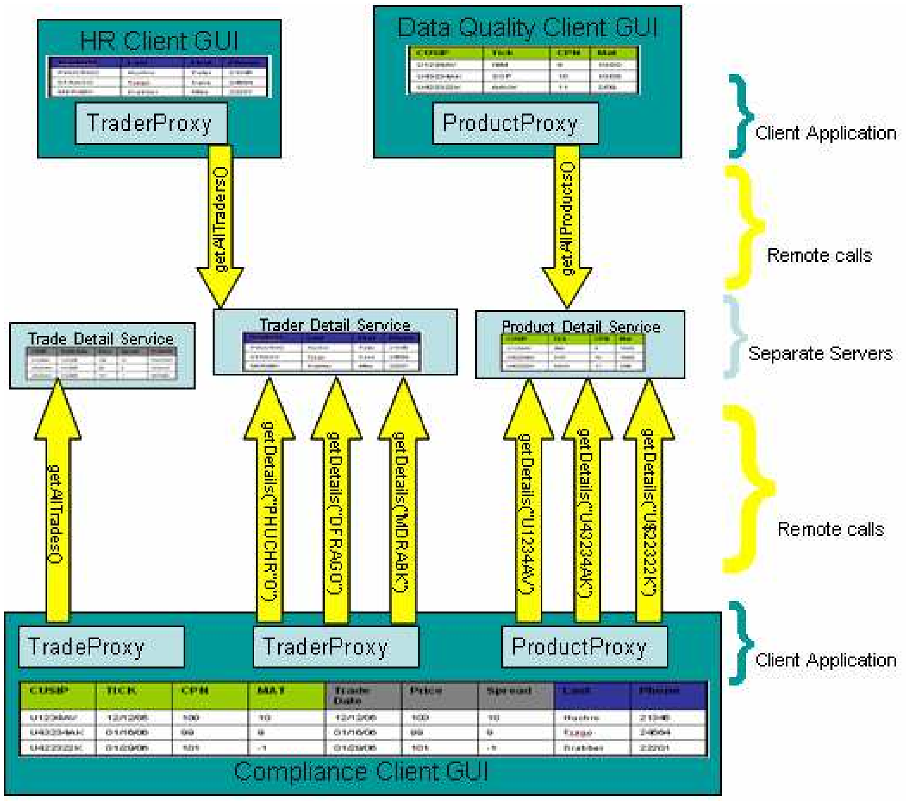

Figure 1 Example Environment

With this design implemented in NOPL, three separate groups can remotely access common functions via a common interface with the "Remote calls" abstracted by the NOPL language.[4]

Goals

Network Abstraction

The primary goal of NOPL is to abstract the concept of the network from functional programs. Instead of writing socket based code and defining a protocol for client/server communication, developers define functions with the keyword *network* to make it available in a distributed environment. The following NOPL Hello World program compiled with the NOPL compiler generates a .java file that when compiled into java byte code and executed, listens on a port for requests to call this function.

[4] See the Example Code section for a NOPL implementation of this specification.

```
HelloServer {
            network String helloWorld(String _name) {
                  return "Hello " + _name; }
}
```

Easy Access over a Network

To abstract the network from the client, the NOPL compiler generates a Java proxy that other programs use to call a function. The only hint at network interaction is the Proxy exception the method may throw. The generated API is:

```
public class HelloServiceProxy {
            private HelloServiceProxy() { // private for Singleton impl };
            public HelloServiceProxy getInstance() { // implement as a singleton };
            public static String sayHello(String _name) throws ProxyException { // impl; };
}
```

And a client uses it as follows:

```
public class MyProgram{
            public static void main(String _args[])  {
              try {
                          String out = HelloServiceProxy.sayHello("Dan");
                          System.out.println(out);
                   } catch (ProxyException pe) {
                          //…
                   }
              }
            }
}
```

Responsive/Fault Tolerant

The inherent slowdown of making a remote call over a network verses a local method invocation takes away from the primary goal of network abstraction. To minimize the impact the network has on execution speed, the designer chose UDP as the underlying transport protocol. In general, UDP is the optimal choice since the client and server do not spend time handshaking, dealing with congestion control, or reliability.[5] To compensate for the unreliability of UDP though, the generated proxy classes have basic retry logic.

Easy to Learn

NOPL is a functional language that implements a few basic data types (string, int, double, float, and date,) a few control structures (if/else and for,) a few collection constructs (enumeration and array,) and some basic mathematical operators.

Easy to Optimization

Since the network is totally abstracted from the server and client developer, the designers can change in the internals of the language to optimize or enhance its behavior without having to change client code. If the designers feel that UDP is too unreliable, he can change the underlying implementation to TCP or even further to a MOM like TIBCO®.

[5] For a detailed understanding of why UDP is faster than TCP see [Kurose et al 2005 pp 196-198.]

Language Caveats:

- Since the client program communicates with the server via a Java proxy, the client programmer must remember that any objects the proxy returns are passed by value rather than by reference. Specifically, changes made on the client side are NOT reflected on the server side.
- Without an object request broker (ORB) or naming service (JNDI) in place, client code must have *a priori* knowledge of the server location (IP address) and port. Clients can implement extendable solutions by setting such parameters in the Java properties at run time. Future enhancements to the language can include usage of a naming service similar to Java RMI.[6]

Example Code

This is code written in NOPL.

```
/** Code written in NOPL*/
TraderDetailsService  {
        networked  Trader[] getDetails(String _traderName) {  // implementation };
        networked  Trader[] getAllTraders() {  // implementation };
}
```

This is the output of the NOPL compiler (the server.)

```
/**
 * Server Output of the NOPL Compiler
 */
public class TradeDetailsService {
        private Object[] getDetails(String _traderName) {  // implementation };
        private Object[] getAllTraders() {  // implementation };
        public static void main(String _args) {
                // Start listening on a specified port and direct
                // calls to either getDetails or getAllTraders.
                // loop ad infinitum
        }
}
```

This is the output of the NOPL compiler (the client.)

```
/**
 * Client generated java stub
 */
public class TradeDetailsServiceProxy  {
        private TradeDetailsServiceProxy () { // private for Singleton impl };
        public TradeDetailsServiceProxy getInstance() { // implement as a singleton };
        public static Object[] getDetails(String _traderName) throws ProxyException
        { // implementation makes a remote call; }
        public static Object[] getAllTraders() throws ProxyException
        { // implementation makes a remote call; }
}
```

References

Kurose, Jim & Ross, Keith: Computer Networking. A Top-Down Approach Featuring the Internet. Third Edition Addison Wesley, 2005

Sun Microsystems Inc. RMI Optional Package (RMI OP); JSR 66 Overview. Feb 2006. Online. Internet http://java.sun.com/products/rmiop/overview.html

[6] See http://java.sun.com/products/rmiop/overview.html for a detail description of RMI and JNDI.