



JNBridgePro™ Performance Hints

www.jnbridge.com



Introduction

The JNBridgePro Java/.NET interoperability tool offers a great deal of flexibility in accessing Java classes from .NET. However, depending on the way those classes are accessed, and how the application is architected, the performance overhead of using JNBridgePro can range from nonexistent to noticeable. This document provides a number of hints that can help to improve JNBridgePro performance. As new performance hints become available, they will be added to this document. If you have a performance hint that's not mentioned here, please mail it to support@jnbridge.com.

The hints included in this document are:

1. Use the binary communications protocol
2. Upgrade your version of JNBridgePro
3. Place the Java-side component inside the J2EE application server
4. Place the Java-side component on the same machine as the .NET-side component
5. Reduce the number of round trips
6. Return arrays instead of using iterators or enumerations
7. Return arrays instead of repeatedly requesting new values
8. Use value objects
9. Use directly mapped collections

1. Use the binary communications protocol

The binary/TCP communications mechanism supported by JNBridgePro is over an order of magnitude faster than the other supported communications mechanism, SOAP/HTTP. The only time that SOAP/HTTP should be used is when communications between the .NET and Java sides is over the Internet, and the communications must traverse firewalls which block passage of generalized TCP communications.

To use the binary/TCP communications channel, make sure that the `jnbproxy.config` remoting configuration file is set up to use the `jtcp:` protocol instead of the `http:` protocol. Also, make sure that the Java-side configuration file `jnbcore.properties` has the `servertype` property set to `tcp`, not `http`. See the *Users' Guide* and the *Installation Guide* for more details.

2. Upgrade your version of JNBridgePro

If you're using an older version of JNBridgePro, you should consider upgrading to a new version. Each successive version contains optimizations and feature enhancements that improve performance.

To take advantage of these improvements, download and install the latest version, then re-generate your proxies using the new version's proxy generation tool. Also, replace the old copies of `jnshare.dll` and `jnbcore.jar` in your application with copies of the new files that come with the new version.

3. Place the Java-side component inside the J2EE application server

If you're using JNBridgePro to access Enterprise Java Beans running inside a J2EE application server, deploy the JNBridgePro Java side (the war file containing `jnbcore.jar`, the jar files containing the EJB stubs, and the associated configuration files) inside the application server, too. (See Figure 1.) By doing this, `jnbcore.jar`, which is actually doing the EJB calls, gets to bypass RMI (Java Remote Method Invocation) and make direct calls to the EJBs. In fact, placing the Java component inside the application server and having a .NET client access it and the EJBs using JNBridgePro is faster than accessing the EJBs from a Java client over RMI.

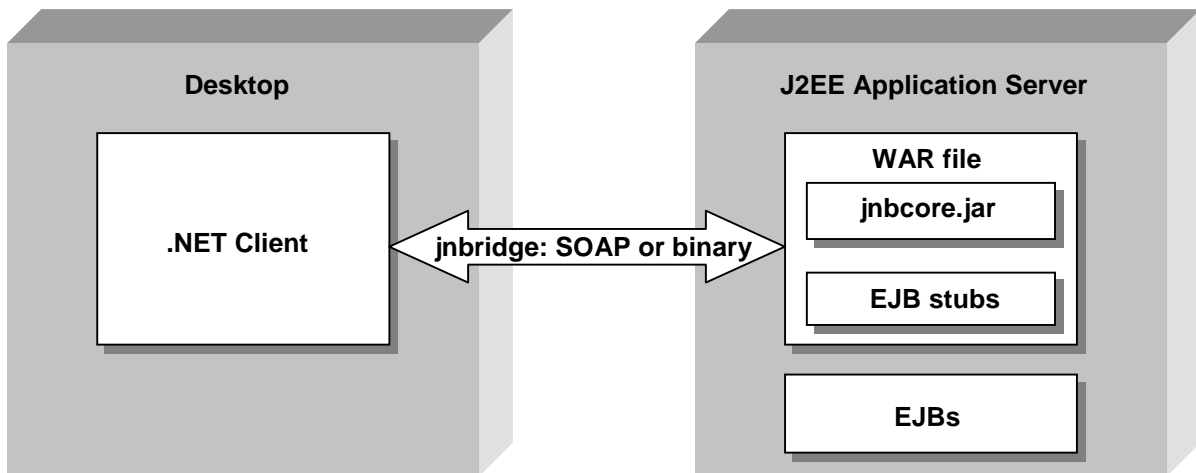


Figure 1: Java-side component in J2EE application server

4. Place the Java-side component on the same machine as the .NET-side component

If the Java classes are being accessed from a single .NET client, the Java-side component, including the Java classes, should be placed on the same machine as the .NET-side component, if possible. This way, the overhead of network communication is avoided.

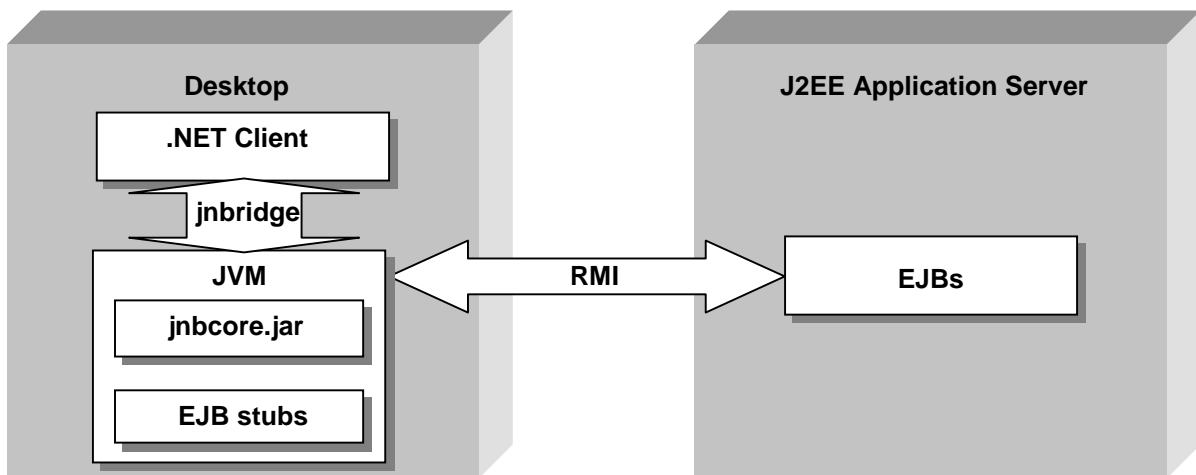


Figure 2: Java-side component on the .NET machine

5. Reduce the number of round trips

One of the best ways to improve JNBridgePro performance is to reduce the number of round trips from the .NET side to the Java side and back, essentially by reducing the number of calls to proxies. Many of the later suggestions in this document are specific suggestions on how to reduce the number of round trips, but one general suggestion is: when a number of proxy calls are necessary to obtain information from which another value is calculated, reduce the number of proxy calls by creating a Java façade class that performs the calls, does the calculation, and returns the value. Then, create a proxy for this class. The value can now be obtained through a single proxy call.

As an example, consider the following C# code fragment. `p1`, `p2`, and `p3` are proxies representing Java objects.

```
x = p1.a(); // round-trip call
y = p2.b(); // round-trip call
z = p3.c(); // round-trip call
w = f(x, y, z); // local .NET call
```

If we create a Java-side façade class that calls `p1`, `p2`, and `p3`, and then performs the calculation `f()`, we can create a proxy for the façade class and get this value with a single round-trip call:

```
w = facadeProxy.getValue();
```

Note that this is only practical if you have access to the Java code and can add that class.

6. Return arrays instead of using iterators or enumerations

JNBridgePro offers full access to Java APIs, and there is a great temptation to make full use of them. For example, if there is a Java-side `Vector` object, one may wish to iterate through it on the .NET-side by creating proxies for `java.util.Vector`, `java.lang.Object`, and `java.util.Enumeration` and then write the following C# code:

```
// let v be a java.util.Vector proxy object
for (Enumeration e = v.elements(); e.hasMoreElements(); )
{
    Object o = e.nextElement();
    // do something with o
}
```

The problem is that each iteration through the loop involves two round-trip calls (to `e.hasMoreElements()` and `e.nextElement()`, in addition to whatever is done with `o`). An alternative approach that avoids these round trips is to extract an array from the `Vector` and to iterate through the array. Since arrays are returned by value from Java to .NET, and are represented natively in .NET, the extra round trips are eliminated:

```
Object[] oArray = v.toArray();
foreach( Object o in oArray )
{
    // do something with o
}
```

7. Return arrays instead of repeatedly requesting new values

Some Java APIs support stateful objects that may be repeatedly called to obtain additional information. The JDBC class `ResultSet`, for example, represents the results of a query, which can contain multiple rows and which must be scrolled through. Consider the following C# code fragment that uses proxies for `ResultSet` and `Statement`:

```

ResultSet rs = theStatement.executeQuery(theQuery);
rs.first();
while(true)
{
    // process the row here

    if (rs.isLast()) break;
    rs.next();
}

```

For each row processed, there are at least two round-trip calls (`isLast()` and `next()`), and the code that processes the row will also contain round-trip calls to the `ResultSet` object. These multiple round-trip calls to `ResultSet` can be a large performance hit.

To improve performance, one can create a Java wrapper class that returns an array of objects each containing one row of results. Assuming that `wrappedStatement` is an instance of the just-described wrapper class that performs the query, accesses each row, gets the data in each row and constructs a `Result` object, and returns an array of all `Result` objects, we can use the following code:

```

Result[] results = wrappedStatement.executeQuery(theQuery);
foreach (Result r in results)
{
    // process r here
}

```

In this case, we avoid the round-trip calls to `first()`, `isLast()`, and `next()`. In addition, if `Result` is a value object (see below), then any accesses of its fields will be executed on the .NET side and will avoid the round trip.

If there is a chance that the results array is very large, and that returning its entire contents will take a long time, one can modify the wrapper class to return a limited number of results at one time (for example, no more than 50).

8. Use value objects

By default, objects returned from .NET calls to Java methods are passed by reference. That is, the object continues to reside on the Java side, and only a reference (a way to find the object) is returned. References are generally much smaller than the actual object, and therefore much faster to return, but to get any useful data out of the object, one must go back to the Java side to get the data. Thus, each data access, even if it is only to get the value of a field, requires a round trip. If all you are doing to such an object is looking up data in fields (or, if it is a Java Bean, if you are looking up data through accessor methods), and you are doing this a lot, it makes sense to designate objects of that class as value objects. A value object can be thought of as a snapshot of an object created on the Java side and copied back to the .NET side. Depending on the kind of value object it is, either the values of its public fields will be copied, or the values of its accessor (`get`) methods (in the case of a Java Bean value object). The object's methods (other than `get` methods for a Java Bean value object) are not copied, since it's problematic to automatically translate the meanings of Java methods to .NET.

It's worthwhile to use value objects when the object is really just a large package of data that can be accessed in lots of different ways. For example, an object representing a customer's bank account might have fields for the customer's account number, first name, last name, address, current balance, previous balances, and the date the account was established. If `accountCollection` is a proxy of a Java object containing user accounts, C# code to print the user's account information might look like this:

```

Account userAcct = accountCollection.getAccountByID(idNumber);
System.Console.WriteLine("ID = " + userAcct.id);
System.Console.WriteLine("First name = " + userAcct.firstName);
System.Console.WriteLine("Last name = " + userAcct.lastName);
System.Console.WriteLine("Address = " + userAcct.address);
System.Console.WriteLine("Current balance = " + userAcct.currentBalance);
foreach(int previousBalance in userAcct.previousBalances)
{
    System.Console.WriteLine("PreviousBalance = " + previousBalance);
}
System.Console.WriteLine("Account date = " + userAcct.accountDate);

```

If the `Account` object `userAcct` is passed by reference, which is the default, then each field access in `userAcct` is a round trip. However, if `Account` is passed by value, then the data in `userAcct` is automatically copied from the Java side to the .NET side and each field access is local.

Whether it is advisable to pass an object by value or by reference depends on the size of the object (which determines how long it takes to be passed from Java to .NET) and how much of its data will be accessed. For example, in the `userAcct` example above, if the only data being accessed in `userAcct` is the current balance, then it might not be worthwhile to pass `userAcct` by value, since the time taken to copy all the data to the .NET side may outweigh the time savings that result by making all field accesses local to .NET. In such cases, the decision on whether to pass the object by value or by reference depends on measurement and the developer's judgement.

Value objects may also be useful when passing objects as parameters. For example, to create a new bank account using the same system described above, the following C# code (using the proxy objects `userAcct` and `accountCollection`) might be used:

```

Account userAcct = new Account();
userAcct.id = acctID;
userAcct.firstName = firstName;
userAcct.lastName = lastName;
userAcct.address = address;
userAcct.currentBalance = currentBalance;
userAcct.previousBalances = new int[0];
userAcct.accountDate = accountDate;
accountCollection.addNewAccount(userAcct);

```

If `userAcct` is a reference object, then each field assignment is a round trip (as is the constructor), but if `userAcct` is a value object, then object construction and assignment of fields is all local. Once the object is set up, it is then copied to the Java side through the parameter mechanism.

In both the above examples (returning an object, and passing one as a parameter), the code is the same regardless of whether the object is passed by reference or by value; the only difference is whether the object's class is designated as a value object or a reference object.

It should be noted that individual objects cannot be designated as pass-by-reference or pass-by-value; all objects of a given class have the designation. For more information on value objects and reference objects, see the Users' Guide.

9. Use directly mapped collections

Directly mapped collections are a way to return certain object collections from the Java side, and have them automatically converted to native .NET collections on the Java side. After they are converted, their elements can be quickly accessed, without a round trip. The example given above ("Return



arrays instead of using iterators or enumerations”) which improves the performance of iterating over the elements of a Java vector, can be rewritten using directly mapped collections:

```
// v is a proxy object for java.util.Vector
System.ArrayList al = v.NativeImpl;
foreach( Object o in al )
{
    // do something with o
}
```

If the proxy for `java.lang.Vector` is designated as a directly mapped collection, then it is a thin wrapper for a `.NET ArrayList` which is accessed through the `NativeImpl` property. Accesses to the `ArrayList` are fast because they are local to `.NET` and do not involve round trips.

JNBridgePro supports a variety of directly mapped collections. Java Vectors, ArrayLists, LinkedLists, and HashSets are directly mapped (to `.NET ArrayLists`), as are Java Hashtables and HashMaps (to `.NET Hashtables`). It is also possible to use directly mapped collections to pass parameters from `.NET` to Java. See the *Users' Guide* for more information.

As with value objects, directly mapped collections take longer to pass between Java and `.NET` than reference objects, but accessing their elements is faster. Deciding whether to use a directly mapped collection depends on the size of the collections being transferred, the number and frequency of accesses, and the developer's judgement.

Each class whose functionality is to be exposed will cause a proxy of the same name to be generated. The generated proxy's members (including constructors, methods, and fields) will correspond to the members of the Java class underlying the proxy.

COPYRIGHT © 2002-2005 JNBridge, LLC. All rights reserved. JNBridge is a registered trademark and JNBridgePro and the JNBridge logo are trademarks of JNBridge, LLC. Java is a registered trademark of Sun Microsystems, Inc. in the United States and other countries. Microsoft, Windows, and other Microsoft product names are trademarks or registered trademarks of Microsoft Corporation in the United States and other countries. Other terms and product names may be trademarks or registered trademarks of their respective owners and are hereby acknowledged.

November 1, 2005