NASA/TM—2001-210884

Enhancing the Remote Variable Operations in NPSS/CCDK

Janche Sang
Cleveland State University, Cleveland, Ohio

Gregory Follen, Chan Kim, Isaac Lopez, and Scott Townsend
Glenn Research Center, Cleveland, Ohio

Prepared for the
2001 International Conference on Parallel and Distributed Systems
cosponsored by the Korea Information Science Society and the
Institute of Electrical and Electronics Engineers
KyongJu City, Korea, June 26–29, 2001

National Aeronautics and
Space Administration

Glenn Research Center

May 2001

Acknowledgments

The authors would like to express their appreciation to management of the High Performance Computing and Communications Program for supporting NPSS.

Enhancing the Remote Variable Operations in NPSS/CCDK

Janche Sang
Department of Computer and Information Science
Cleveland State University
Cleveland, Ohio 44115

Gregory Follen, Chan Kim, Isaac Lopez, Scott Townsend
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

Abstract

Many scientific applications in aerodynamics and solid mechanics are written in Fortran. Refitting these legacy Fortran codes with distributed objects can increase the code reusability. The remote variable scheme provided in NPSS/CCDK helps programmers easily migrate the Fortran codes towards a client-server platform. This scheme gives the client the capability of accessing the variables at the server site. In this paper, we review and enhance the remote variable scheme by using the operator overloading features in C++. The enhancement enables NPSS programmers to use remote variables in much the same way as traditional variables. The remote variable scheme adopts the lazy update approach and the prefetch method. The design strategies and implementation techniques are described in details. Preliminary performance evaluation shows that communication overhead can be greatly reduced.

1 Introduction

A distributed object is a reusable, self-contained piece of software that cooperates with other objects on the same machine or across a network in a plug-and-play fashion via a well-defined interface. The Numerical Propulsion System Simulation (NPSS) [7] attempts to provide a collaborative design and simulation environment based on this concept. It uses object-oriented technologies such as C++ objects to encapsulate individual engine components and Common Object Request Broker Architecture(CORBA) [8] for object communication and deployment across heterogeneous computing platform. Many scientific applications in aerodynamics and solid mechanics are written in Fortran. Refitting these legacy Fortran codes with distributed objects can increase the code reusability. The NPSS provides a CORBA Component Development Kit (CCDK) which is a C++ wrapper library. It is easier for programmers to code using the CCDK than to implement all the functionality using CORBA IDL.

Because CORBA IDL to Fortran mapping has not been proposed, there seems to be no direct method of generating CORBA objects from Fortran without using CORBA/C++ wrappers. The wrapper is responsible for accepting the request from the clients and then launching the encapsulated legacy codes. Based on the level of encapsulation, we can classify the wrapping approaches into two broad categories: shell script

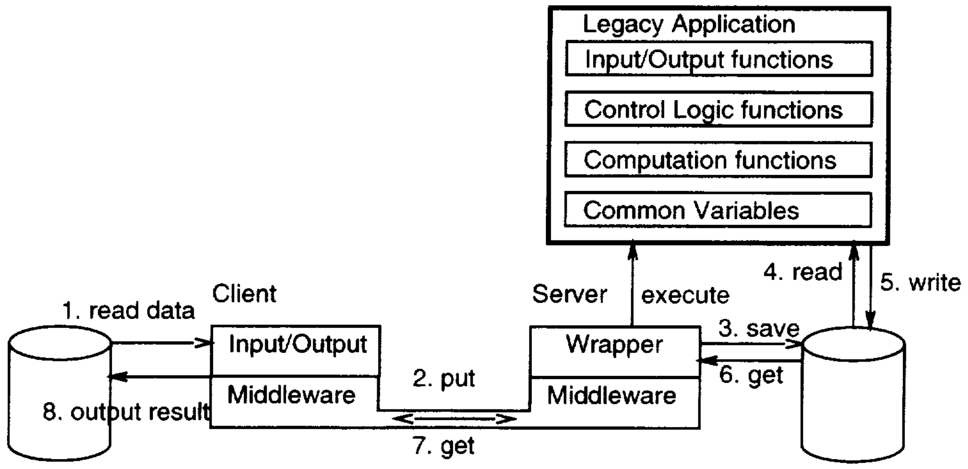

Figure 1: Script Wrapping

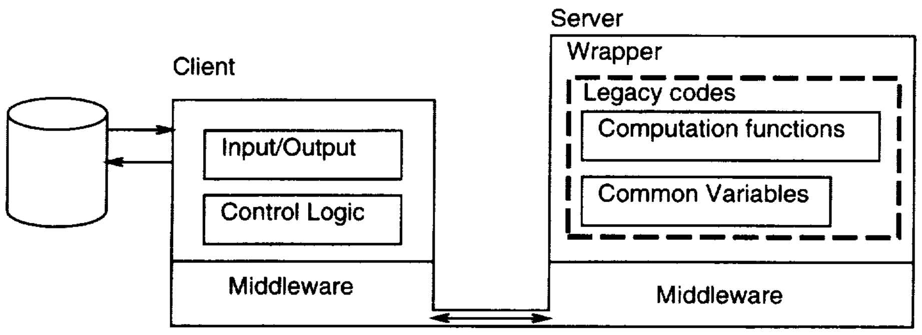

Figure 2: Re-Structure Wrapping

wrapping and re-structure wrapping. The shell script wrapping approach involves writing wrappers that provide the input data for the code, execute the code as a shell command (e.g. using the UNIX`system()` function), and retrieve the output result (see Figure 1). That is, the wrapper performs all of its interactions with the legacy code via input and output files. The major advantage of this approach is that programmers do not need to modify the Fortran code.

The re-structure wrapping approach involves re-engineering the Fortran code and compiling/linking the code with CORBA/C++ wrapper. Namely, the code is re-organized and partitioned into client and server parts (see Figure 2). Generally, a scientific application contains I/O module, control logic module, and computation module. Using this approach, the control logic module could be moved to the client site to give the client the flexibility of controlling and invoking individual iteration used in most scientific computation.

Note that the computation functions at the server site can be further decomposed into several objects or components. A semi-automatic conversion tool which takes the Fortran program as input and helps programmers generate a C++ header file and an IDL file for wrapping can be found in [13]. Strategies about program decomposition and object extraction have been discussed in references [1], [9], and [15]. This topic is beyond the scope of the paper.

Both of the wrapping approaches should deal with the issues of passing the input data and getting the result between the client and the server. For the shell script wrapping approach, the wrapper needs to transfer the input/output data files. A feasible implementation method is to read the file into a multi-string format variable, pass the variable to the remote site through a function argument, and restore/save the variable to a file. For the re-structure approach, one method described in [13] extracts the Fortran COMMON blocks and converts them into a structure-type attributes in CORBA IDL. Through the attributes, the client can initialize the variables which will be used by the server.

An alternative re-structure method supported by the NPSS/CCDK is based on the remote variable scheme. The server should register the variable with a name as the key to allow the client to access. The client can set and get the variable's value through it's registered name. In this research, we are interested in the enhancement of the remote variable programming interfaces in NPSS/CCDK. Our objective is to define the remote variable operators to mimic conventional variable usage. With the support of remote variable operations which have the same notations as traditional variables, the programmers can easily migrate their Fortran codes towards a client-server platform.

2 The Remote Variable Scheme in NPSS/CCDK

NPSS/CCDK supports a remote variable scheme between the client and the server[16]. This scheme is based on the distributed shared memory (DSM) technology in which data are logically shared but physically distributed over networks of workstations. Depending on the level of implementation, we can classify the approaches to supporting DSM into two major categories: page-based and object-based. The former is usually implemented at the operating system level through an extended memory manager[5]. Transparency is the major advantage of this approach because the memory system is totally hidden from users. The object-based approach is often realized at the language or the library level to support shared objects[6]. It is more flexible and requires less development efforts than the system dependent page-based approach.

The NPSS/CCDK remote variable scheme is a simplified version of the object-based DSM mechanism. As we mentioned earlier, the purpose of this scheme is to provide a simple and elegant data-sharing capability in the client-server platform. The client can read data into the shared variables which will be used by the server to perform some computations. Then, the client can get the variables' values back to either output the result or to issue another request of iteration. That is, the variables are shared only between the client and the server processes in a sequential manner. Therefore, there is no immediate need to support complicated synchronization primitives and memory consistency policies addressed in the DSM technology.

Figure 3 depicts the framework of the remote variables in NPSS/CCDK. When the server starts, each shared variable should be registered with a character-string name as the key. For example, a programmer can use the following Fortran statement to register the real variableX:

```
        call NPSS_registerReal4Ref('X', X, NPSS_INPUT, 'units', 'description', err)
```

Note that Fortran adopts call-by-reference as the parameter passing scheme. Hence, the location of the shared variable X is passed to the registration function via the second parameter. This address is stored along with the key in the shared variable list. To declare a corresponding remote variableX at the client site, the following statement is used:

```
        RemoteReal X = RemoteReal(cspan, "X");
```

As shown in Figure 3, each remote variable should be bound to a particular server which hosts the variable.

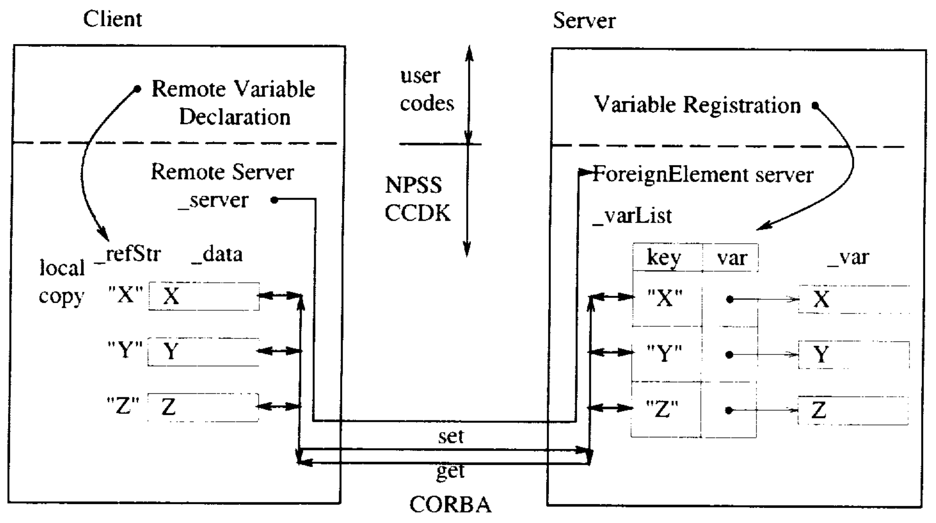

Figure 3: Framework of the NPSS/CCDK Remote Variables

The character-string name should be provided during the declaration in order to find the variable's location from the variable list at the server.

Two basic functions `set()` and `get()` are provided for setting and getting the remote variable's value, respectively. Figure 4 shows in details the implementation of the`set()` and `get()` methods which belong to the `ForeignElement` server class. As described in [16], the `ForeignElement` class implements the `CorbaExtElement` interface and provides an API with which the legacy Fortran code can be connected. The calling sequence $2 \longrightarrow 3 : 6 \longrightarrow 8$ in Figure 4 shows an example of such connection between Fortran and the ForeignElement class. The symbol $\longrightarrow$ represents a function call, while the symbol : denotes an execution sequence. That is, the registration of the remote Fortran variable X at line 2 calls the C++ function `npss_registerreal4ref__` at line 3. After executing a few statements, this function invokes the ForeignElement method `registerReal4Ref` at line 6 and continues its execution at line 8.

The registered variable is inserted into the variable list`_varList` in the `ForeignElement::registerReal4Ref` method (at line 10) and is being searched via`lookup()` in the `set()` and `get()` methods (at line 16 and 24). Below we use α and β to denote the server site's `set()` and `get()` method invocation sequences with their line numbers as shown in Figure 4.

$$\alpha \quad 21 : 25 \longrightarrow 37 : 39 \longrightarrow 49$$

$$\beta \quad 13 : 17 \longrightarrow 32 : 34 \longrightarrow 55$$

The CORBA-based `ForeignElement` class is encapsulated by another family of classes which include `RemoteServer`, `RemoteVariable`, `RemoteNumber`, etc. These classes provide user-friendly interfaces to the client and their implementation can be found in Figure 5. The calling sequences of the`set()` and `get()` methods at the client are shown below:

$$\gamma \quad 37 : 41 \longrightarrow 55 : 58 \longrightarrow \alpha$$

$$\delta \quad 22 : 24 \longrightarrow 50 : 53 \longrightarrow \beta$$

Note that the method invocations at line 58 and 53 will start the sequences α and β at the server, respectively. To thoroughly understand how the remote variable scheme works between the client and the server, interested readers can follow these four method invocation sequences to trace NPSS/CCDK source codes.

3 Enhancement of the Remote Variable Operations

3.1 Operator Overloading

With the support of the `Foreign`-related and `Remote`-related families of classes in CCDK, programmers can easily re-engineer their legacy Fortran programs into the client-server platform. The efforts of learning and programming with CORBA can be totally reduced. However, there are some rooms for improvement in the remote variable operations. One is the asymmetrical relationship between getting and setting the remote value. "Get" is done via indirection (unary '*') while "Set" is done by normal assignment. Another is the lack of index operator (i.e. [ ]) support for the remote array element access. Programmers need to call specific functions such as `setElement()` or `getElement()` to access remote array elements. Figure 6 shows these two problems.

NPSS programmers have to be aware of these uncommon operations on remotely based data. To alleviate programmer's burden, we need to modify the `Remote-` family of classes. In the old implementation, the reason why an unary '*' is needed is to find the r-value of the variable. It distinguishes that its position is on the right-hand side of the assignment operator `=`. We find that this can be replaced with the ADT type-conversion overloading operator

```
        operator type() { ...  }
```

because if the variable appears in an expression, this type-conversion function will be implicitly invoked[10]. As shown in Figure 7, the overloaded `operator T()` invokes the method `get()` and returns the variable's value.

To encapsulate the functions `setElement()` and `getElement()` of remote-array access, we also overload the index operator `[ ]` (see Figure 8). The old implementation lets one local object represent all of the array elements. A problem arises when programmers use the following statement

```
        ARO[0] = ARO[1];
```

the first index 0 will be overwritten by the second index 1 because they share the same space in the local object. To solve this problem, we add one more level of the remote array class. The intermediate class `_RemoteArray1D` represents an individual array element and each has its own index number. The higher level class contains an array of the intermediate class elements and an overloading index operator`[ ]`. Note that our current implementation allocates array elements via `new` inside a for loop. The performance cost is very high. Future modification is needed to have a one-time allocation instead of requesting element space one by one.

The `cin` and `cout` operators are also overloaded for the remote variables. Therefore, programmers can read data into remote variables using these operators. Figure 9 shows the new usage. In fact, these overloading operators for remote variables allow users to write client-server programs just as using traditional languages.

```
-------------------------------------- npss_int.f ---------------------------------------------
c      Register namelist variables.
       call NPSS_registerReal4Ref('X', X, NPSS_INPUT, 'units', 'description', err)
-------------------------------------- Fortran.C ----------------------------------------------
void npss_registerreal4ref__ (const char* name, NCPReal4& value, int& ioStatus, ...)
{ ForeignElement* self = ForeignElement::self();
  ...
  self->registerReal4Ref(nameStr, value, IOS(ioStatus), unitStr, descStr);
}
-------------------------------------- ForeignElement.C ---------------------------------------
void ForeignElement::registerReal4Ref(const char* name, NCPReal4& value,...)
{ CVariableBasePtr newVar = new CFloatRef(name, value);
  _varList.insertKeyAndValue(&(newVar->getName()), newVar);
  registerAttributes(newVar, ioStatus, units, description);
}
CORBA::Any* ForeignElement::get(const char* name ADD_ENV_DECL)
{ CORBA::Any* anyPtr;
  NCPString attrName;
  CVariableBasePtr it = lookup(name, attrName);
  anyPtr = it->get(attrName);
  ...
  return anyPtr;
}
void ForeignElement::set(const char* name, const CORBA::Any& value,CORBA::Boolean restricted ADD_ENV_DECL)
{ ...
  NCPString attrName;
  CVariableBasePtr it = lookup(name, attrName);
  it->set(attrName, value, restricted);
}
CVariableBasePtr ForeignElement::lookup(const char* name, NCPString& attrName)
{  ...
   CVariableBasePtr it = _varList.findValue(&findName);
   return it;
}
-------------------------------------- CCDK/CBase/CVariableBase.C ------------------------------
CORBA::Any *CVariableBase::get(const char*name){
  CORBA::Any *it=new CORBA::Any();
          getToAny(name,*it);
   ...
}
void CVariableBase::set(const char *name,const CORBA::Any &it,CORBA::Boolean restricted){
   ...
   setFromAny(name,it);
}
-------------------------------------- ForeignVariable.H --------------------------------------
class CFloatRef : public CVariableBase {
  protected:
    NCPReal4& _var;
    ...
  public:
    CFloatRef(const char* name, NCPReal4& it) : CVariableBase(name), _var(it) {};
    ...
}
-------------------------------------- ForeignVariable.C --------------------------------------
void CFloatRef::setFromAny(const char* name, const CORBA::Any& it)
{ ...
   if (it >>= doubleVal) {
      _var = doubleVal;
  ...
}
void CFloatRef::getToAny(const char* name, CORBA::Any& it)
{   ...
    double doubleVal = _var;
    it <<= doubleVal;
}
```

Figure 4: Server code

```
-------------------------------------- Client.C ------------------------------------------------
RemoteServer* cspan;
cspan = new RemoteServer("CSPANdemo");
RemoteReal X = RemoteReal(cspan, "X");
X = 1.25;
-------------------------------------- RemoteVariable.H ----------------------------------------
typedef RemoteNumber<CORBA::Double> RemoteReal;
template <class T> class RemoteNumber : public RemoteVariable<T> {
...
}
template <class T> class RemoteVariable {
  protected:
    // Local copy of the data.
    T _data;

    // The (bound) remote server to use.
    RemoteServer*& _server;

    // Name of variable on remote server.
    RWCString _refStr;
    ...
}
template <class T>
void RemoteVariable<T>::get(void)
{
  CORBA::Any* anyPtr = _server->get(_refStr);
  if (!_get(anyPtr)) {
    ...
  }
}

template <class T>
int RemoteNumber<T>::_get(CORBA::Any* anyPtr)
{
  return *anyPtr >>= _data;
}

template <class T>
void RemoteVariable<T>::set(void)
{ CORBA::Any anyVal;

  anyVal <<= _data;
  _server->set(_refStr, anyVal);
}
-------------------------------------- RemoteServer.H ------------------------------------------
class RemoteServer {
 ...
 private:
   NCPCorba::CorbaExtElement* _server;
 ...
}

-------------------------------------- RemoteServer.C ------------------------------------------
CORBA::Any* RemoteServer::get(const char* varName)
{ CORBA::Any* anyPtr;
  ...
  anyPtr = _server->get(varName);
}
void RemoteServer::set(const char* varName, CORBA::Any& value,
                       int synchronous)
{  ...
   _server->set(varName, value, 1);
}
```

Figure 5: Client code

```
Server:
   call NPSS_registerReal4Ref('CP',CP,NPSS_INPUT,'units','description',err)
   call  NPSS_registerReal4Array1Dref('ARO',ARO,50,NPSS_INPUT,'units','description',
err)

Client:
   RemoteReal CP = RemoteReal(cspan,"CP");
   RemoteRealArray1D ARO = RemoteRealArray1D(cspan, "ARO");
   CP = 4.0;
   double g;
   g = *CP;  // problem 1
   ARO.setElement(0,3.0); ARO.setElement(1,2.5); // problem 2
   ARO.setElement(2, ARO.getElement(0)+ARO.getElement(1));
   g= ARO.getElement(1);
```

Figure 6: Problems of Uncommon Remote Variable Operations

3.2 Lazy Update and Prefetch

Generally, we can separate the NPSS client-server computing into three phases: the pre-execution phase at client, the server execution phase, and the post-execution phase at client. The remote variables are usually initialized during the pre-execution phase and their computed values are either displayed or used for determining another iteration during the post-execution phase. In the pre-execution phase, many variables can be set. To reduce the number of message transmitted, we adopt the lazy update approach to postpone sending the set requests. These variables' values will be merged, transfered, and set via invoking the function `setMultiple()` just before launching the server's computation. Similarly, when the server execution phase ends, the client can prefetch a few variables through the function`getMultiple()` before these variables are actually used. When these variables are referenced later, their data are available locally. Figure 10 and 11 demonstrates the difference between the immediate streaming requests and the lazy/prefetch approaches with aggregated data.

In our implementation, the pools`W pool v` and `R pool u` declared in the `RemoteServer` class are used to keep track of which variables needs to be lazily set and prefetched, respectively (see Figure 12). To avoid searching the pools to determine whether a variable is in the pools or not, we use two booleans`inWpool` and `inRpool` for each variable to denote its status. A variable will be inserted into the`W pool v` when its value is updated (e.g. `cin >> X;` , `X = 2.0;`). A `flush()` function is provided and will be invoked implicitly before server starts execution.

Since the NPSS CCDK is provided as a library instead of a compiler, there is no way we can determine which variables could be prefetched after the server execution phase finishes. One solution is to let programmers explicitly add a `prefetch()` function call in their `Client.C` program. Our solution is based on the time locality concept used in the cache implementation[14]. If a variable is referenced but the local copy is not fresh, we will get a copy from the server immediately and put this variable in the`R pool u`. Therefore, it can be prefetched in the next iteration. We also use the flag`Referenced` as the reference bit for a variable. It is set to be true when the variable is referenced (see Figure 7 the`operator T()` function). If a variable is in the `R pool u` but not referenced, our current implementation will remove it from the`R pool u`. In fact, it's not necessary to do this because one more variable in prefetching won't cost too much. On the contrary,

```
template <class T> class RemoteVariable: public RV {
  protected:
    T _data;
    string _refStr;
    bool inRpool, inWpool, Referenced;
    bool lazy;// default true, programmers may disable the lazy update
    RemoteServer*& _server;
    ...
    fillzet(NCPCorba::NamedValue & nv) {
         // parameter nv is call by reference
         // called by RemoteServer flush() to merge several var set.
         nv.name = _refStr;
         nv.val <<= _data;
         inWpool = false ;
    }
};

template <class T> class RemoteNumber : public RemoteVariable<T> {
  public:
    RemoteNumber(RemoteServer*& server, const char* refStr)
           : RemoteVariable<T>(server, refStr) {};
    T operator  = (T value) {
        _data = value;
        if (lazy) {
           if (!inWpool) {
                add this object to the W_pool;
                set inWpool to true;
           }
        else
           set(); // send _data back to server immediately
        }
        returen _data;
    }
    operator T() { // rvalue conversion
        if (lazy) {
           if (!inRpool && !inWpool) { // local copy is not fresh
               get(); // get a copy from the server
               add this object to the R_pool; set inRpool to true;
           }
           Referenced = true;
           return _data;
        }
        else
           return get();
    }
    friend istream& operator>>(istream& in, RemoteNumber & x) {
        T temp;
        in >> temp;
        x = temp; // use overloaded =
        return in; // enable cascaded calls };
    ... similar overloading for operators +=, -=, *=, /=, <<  ...
    ...
};
```

Figure 7: Pseudo Codes of the Remote- family of classes

```
template <class T, class E> class _RemoteArray1D : public RemoteStruct<T> {
  public:
    _RemoteArray1D(RemoteServer*& server, const char* refStr, int i)
      : RemoteStruct<T>(server, refStr),indx(i) {};
    E    getElement();
    void setElement(E value);
    operator E() { // rvalue conversion;
        if (lazy) ... similar to operator T()...
        else  getElement();
        return _elm;
    }
    ...
  protected:
    int indx;
    E   _elm; // local copy
};
template <class T, class E> class RemoteArray1D : public RemoteStruct<T>{
  public:
    RemoteArray1D(RemoteServer*& server, const char* refStr)
      : RemoteStruct<T>(server, refStr), aSize(-1),a(0) {}
    _RemoteArray1D<T,E> & operator[](int i)  { // subscript operator,
       if (a == 0) {
           aSize = size(); //  remote method invocation
           a = new _RemoteArray1D<T,E>* [aSize] ;
           for(int i=0; i < aSize; i++) {
              a[i] = new _RemoteArray1D<T,E>(server, refStr,i); //expensive
           }
       }
       if( i >= 0 && i < aSize) {
            return *a[i];
       }
       else ...
    };
    ...
   protected:
     _RemoteArray1D<T,E> **a;
     int aSize;
};
```

Figure 8: Pseudo Codes of the RemoteArray class

```
cin >> CP;
g = CP;
cin>> ARO[0]; ARO[1] = 2.5;
ARO[2] = ARO[0] + ARO[1];
g= ARO[1];

do {
   cspan->execute();
   cout << CP;
   cout <<ARO[2];
} while ( Q < R );
```

Figure 9: Improved Remote Variable Operations

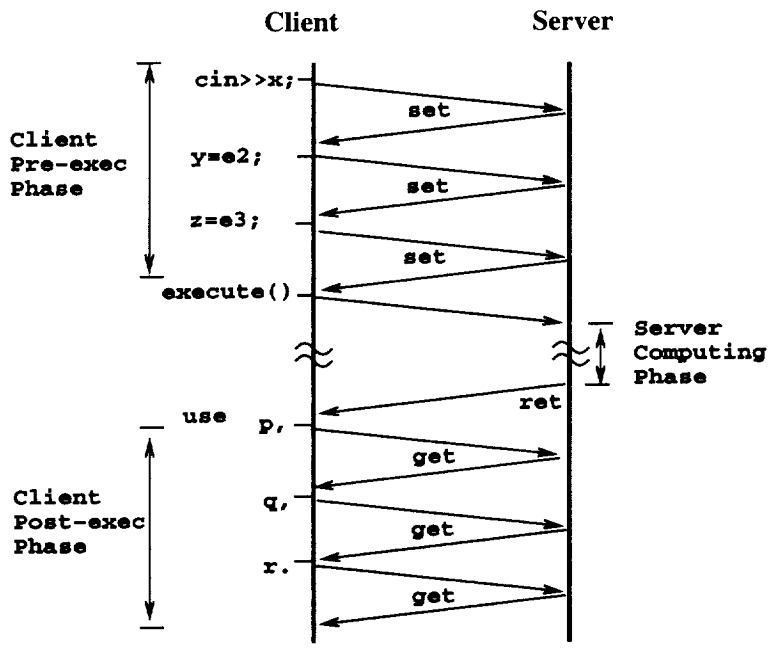

Figure 10: Immediate Set and Get Operations

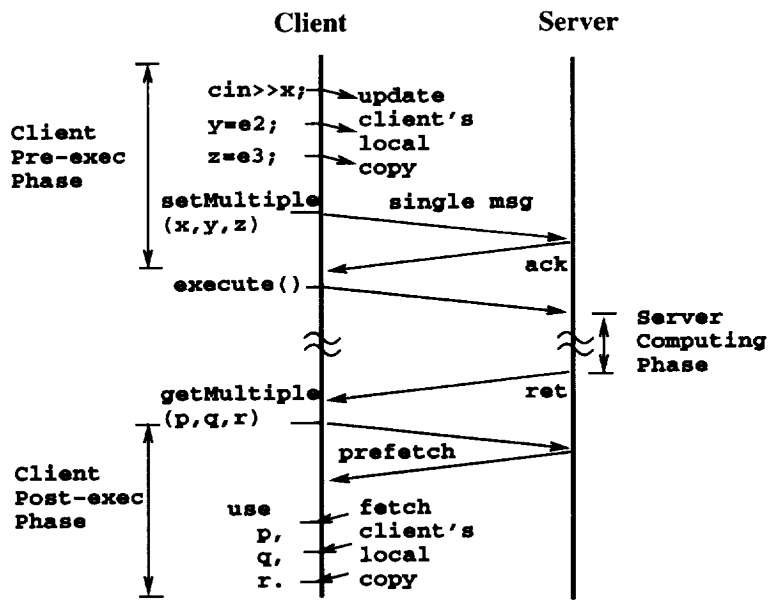

Figure 11: Lazy Update and Prefetch with setMultiple/getMultiple Operations

```
class RV { // an abstract class; inherited by RemoteVariable
 public:
   virtual const char* getName(void) =0;
 friend class RemoteServer; // enable RemoteServer to access protected below
 protected:
   virtual void fillzet(NCPCorba::NamedValue &)  = 0;
   virtual void fillzet(NCPCorba::ANamedValue &)  = 0;
   virtual void fillget(NCPCorba::ANamedValue &)  = 0;
   ...
};
class RemoteServer {
  public:
    void flush(){
        for each object in the W pool v
           put its info (e.g. _refStr, _data) into the aggregated message Ms via fillzet();
           reset its inWpool to false;
        _server->setMultiple(Ms); // invoke a remote method
        clear the W pool v;
        ... similar for array elements pool Av;
    }
    void prefetch() {
        for each object in the R pool u
           if (Referenced == false) { reset its inRpool and remove it from the R pool u);
           else put its info (e.g. _refStr) into the aggregated message Mg via getName();
        Ng = _server->getMultiple(Mg); // get an aggregated message Ng with obj values back
        Decomopose Ng and reset Referenced to false for for each object in the R pool u;
        ... similar for array elements pool Au;
    }
    void execute() { flush(); _server->compute(); prefetch(); }
  private:
    NCPCorba::CorbaExtElement* _server; // pointer to foreign element server
    RWTValOrderedVector< RV *> u;  // R pool for variables
    RWTValOrderedVector< RV *> Au; // R pool for array elements
    RWTValOrderedVector< RV *> v;  // W pool for variables
    RWTValOrderedVector< RV *> Av; // W pool for array elements
};
```

Figure 12: Pseudo Codes of the RemoteServer class

if a variable is referenced in every other post-execution, it's better to keep it in the pre-fetch pool. Like the lazy update approach, the prefetch method is transparent to programmers.

In our implementation, an abstract class called `RV` is created for providing a few virtual functions. This abstract class is inherited by the `RemoteVariable` class (see Figure 8). As shown in Figure 12, the `RemoteServer` class uses the RV class to declare the pools. When the server needs to prepare a merged message for `setMultiple()`, it invokes the polymorphic function `fillzet()` which fills the variable's information into the message buffer.

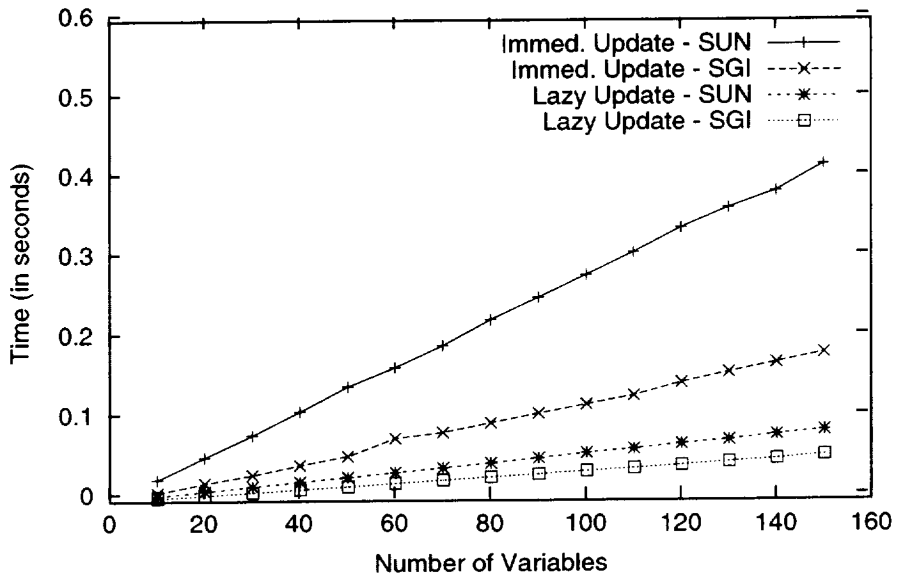

Figure 13: The effect of Lazy Update with setMultiple

4 Performance Evaluation

In this section, we evaluate the performance of the remote variable scheme in NPSS/CCDK. In the first experiment, we evaluate the effectiveness of using the lazy update approach. We measure the elapsed time to update the remote variables for both of the immediate update approach and the lazy update approach. For comparison purpose, we conduct the experiment on two different hardware configurations. One is a pair of Sun Ultra-2 workstations (clock 200MHz, 256MB memory) running Solaris 5.6 over Fast Ethernet[4]. It is worth mentioning that this kind of high-end workstation is fast enough to saturate a 100Mbps Fast Ethernet, The CORBA package Orbix[3] is used in this configuration. while a Sun SPARC 5 (120MHz) cannot[12]. The other configuration is a SGI Origin 2400 with 24 processors (clock 196MHz, 16GB memory) over system inter-connection networks. Because of availability, only the free package MICO[11] is installed on this multiprocessor machine.

Figure 13 shows the elapsed time by varying the number of variables. It is not surprising that the lazy update approach performs much better than the non-lazy approach for both configurations. The effect is more significant over Fast Ethernet than over system bus because the former is slower than the latter. Taking 100 variables as an example, the lazy approach is 4.6 times faster than the non-lazy approach over Fast Ethernet, while it is only 3.1 times faster through system internal connection.

We get very similar performance figures for the prefetch approach with getMultiple. Hence, we don't present them here. Normally, the client may access only a few variables after server execution phase. The prefetch still can gain large performance improvement when the number of iterations increases. For example, assume that there are k variables could be prefetched and n iterations are required. The number of messages will be reduced from $k \times n$ to $k+n-1$. This is because in the first iteration we need to get these k variables, while we only need one single getMultiple request in the rest $n-1$ iterations. Let $t_{get}(x)$ denote the time to get x variables via one message. Then, the variable access time could be reduced from $k \times n \times t_{get}(1)$ to

Benchmark CSPAN	Immed. Update	Lazy Update	Improvement %	Script Wrapping
SUN - Fast Ether	0.405 sec.	0.271 sec.	33%	NA
SGI - System Bus	0.257 sec.	0.202 sec.	21%	0.184 sec

Table 1: Performance Improvement for the Client-Server CSPAN using Lazy Update

$k \times t_{get}(1) + (n-1) \times t_{get}(k)$.

In another experiment, we use the 1-dimensional axial-flow compressor design code CSPAN which is written in Fortran as the benchmark[2]. This code uses a rapid approximate design methodology based on isentropic simple radial equilibrium to determine the flow path either for a given number of stages or for a given overall pressure ratio. We re-engineer the code and migrate it to the client-server platform. For the purpose of comparison, we also implement a file transfer method. The methodputfile we add to NPSS/CCDK takes an input file name as the argument and transfer this file from the client to the server. An example is shown below.

```
    cspan->putfile("fan.input");
```

Table 1 shows the empirical result. Using the lazy update approach, performance can be improved ranging from 21% to 33%. It's interesting that the script wrapping approach performs even better than the lazy update approach.

5 Summary

The remote variable scheme provided in NPSS/CCDK helps programmers easily migrate the Fortran codes towards a client-server platform. The efforts of learning and programming with CORBA can be totally reduced. This scheme gives the client the capability of accessing the variables at the server site. In this paper, we discuss the implementation of the remote variable scheme in CCDK. We identify several areas for improvement and implement them by overloading the data-type conversion operator and the index operator. The enhancement enables NPSS programmers to use remote variables in much the same way as traditional variables. Furthermore, the lazy update and the prefetch approaches used in the scheme greatly reduce the number of messages transmitted. NASA has many research sites. The message transmission cost between two distant sites is usually higher than local. We can expect that NASA researchers will benefit more from using the enhanced NPSS/CCDK when they run the client and server programs across different sites.

References

[1] B. L. Achee and D. L. Carver. Creating Object-Oriented Designs From Legacy Fortran Code. *Journal of Systems and Software*, 39:179–194, 1997.

[2] A. Glassman and T. Lavelle. Enhanced capabilities and Modified users Manual for Axial-Flow Compressor Conceptual Design Code CSPAN. Technical Report 106833, NASA GRC, 1995.

[3] IONA Technologies. URL: http://www.iona.com.

[4] H. W. Johnson. *Fast Ethernet: Dawn of a New Network*. Prentice Hall, Upper Saddle River, NJ 07458, 1996.

[5] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing*, pages 94–101, 1988.

[6] W. Liang, C.T. King, and F.P. Lai. Adsmith: An Object-Based Distributed Shared Memory System for Networks of Workstations. *IEICE Trans. on Information and Systems*, E80-D, No. 9:899–908, 1997.

[7] J. Lytle. The Numerical Propulsion System Simulation: A Multidisciplinary Design System for Aerospace Vehicles, NASA/TM-1999-209194. In *Proceedings of the 14th International Symposium on Air Breathing Engines sponsored by the International Society for Air Breathing Engines*, Sep. 1999.

[8] Object Management Group. *The Common Object Request Broker: Architecture and Specification, 2.3 ed.*, June 1999.

[9] C. Ong and T. Tsai. Class and Object Extraction from Imperative Code. *Journal of Object-Oriented Programming*, pages 58–68, Mar/April 1993.

[10] I. Pohl. *C++ for C Programmers*. Addison-Wesley, Reading, Mass., 3rd edition, 1999.

[11] A. Purder and K. Romer. MICO is CORBA. URL: http://www.mico.org.

[12] J. Sang, C. Kim, T. J. Kollar, and I. Lopez. High-Performance Cluster Computing over Gigabit/Fast Ethernet. *Informatica*, 23, 1999.

[13] J. Sang, C. Kim, and I. Lopez. Developing CORBA-based Distributed Scientific Application from Legacy Fortran Programs. In *Proceedings of the HPCC Computational Aerosciences (CAS) Workshop*, 2000.

[14] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Englewood Cliffs, NJ 07362, 1992.

[15] P. Tonella, G. Antoniol, R. Fiutem, and F. Calzolari. Reverse Engineering 4.7 Million Lines of Code. *Software – Practice and Experience*, 30:129–150, 2000.

[16] S. Townsend. Using External Codes with NPSS . Technical Report in preparation, NASA GRC, 2001.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY (*Leave blank*)	**2. REPORT DATE** May 2001	**3. REPORT TYPE AND DATES COVERED** Technical Memorandum

4. TITLE AND SUBTITLE Enhancing the Remote Variable Operations in NPSS/CCDK	**5. FUNDING NUMBERS** WU-725-10-24-00
6. AUTHOR(S) Janche Sang, Gregory Follen, Chan Kim, Isaac Lopez, and Scott Townsend	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration, John H. Glenn Research Center at Lewis Field, Cleveland, Ohio 44135-3191	**8. PERFORMING ORGANIZATION REPORT NUMBER** E-12753
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration, Washington, DC 20546-0001	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** NASA TM—2001-210884

11. SUPPLEMENTARY NOTES

Prepared for the 2001 International Conference on Parallel and Distributed Systems cosponsored by the Korea Information Science Society and the Institute of Electrical and Electronics Engineers, KyongJu City, Korea, June 26–29, 2001. Janche Sang, Cleveland State University, Department of Computer and Information Science, 1983 E. 24th Street, Cleveland, Ohio 44115-2403; Gregory Follen, Chan Kim, Isaac Lopez, and Scott Townsend, NASA Glenn Research Center. Responsible person, Isaac Lopez, organization code 2900, 216-433-5893.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified - Unlimited; Subject Categories: 07 and 61; Distribution: Nonstandard; Available electronically at http://gltrs.grc.nasa.gov/GLTRS; This publication is available from the NASA Center for AeroSpace Information, 301-621-0390.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 words)***

Many scientific applications in aerodynamics and solid mechanics are written in Fortran. Refitting these legacy Fortran codes with distributed objects can increase the code reusability. The remote variable scheme provided in NPSS/CCDK helps programmers easily migrate the Fortran codes towards a client-server platform. This scheme gives the client the capability of accessing the variables at the server site. In this paper, we review and enhance the remote variable scheme by using the operator overloading features in C++. The enhancement enables NPSS programmers to use remote variables in much the same way as traditional variables. The remote variable scheme adopts the lazy update approach and the prefetch method. The design strategies and implementation techniques are described in details. Preliminary performance evaluation shows that communication overhead can be greatly reduced.

14. SUBJECT TERMS Computer techniques; Parsing algorithms	**15. NUMBER OF PAGES** 21
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT**