Buffered Collection and Buffered Iterator Patterns

from the OOPSLA ‘95 Workshop on Concurrent, Distributed and Parallel Design Patterns.

by Phil Brooks
phil_brooks@mentorg.com
Mentor Graphics Corporation
8005 Boeckman Road
Wilsonville, Oregon 97070

Abstract

This paper describes implementation of a pair of patterns that improve performance of large collections and their iterators in distributed object systems through a combination of buffering and specialized inter-process communication techniques. The Buffered Collection and Buffered Iterator patterns provide a transparent mechanism for efficient transport of primitive data elements between collections residing in one address space and their clients residing in another.

1.0 Introduction

Performance trade-offs between the use of distributed object systems such as CORBA based distributed object services and lower level networking interfaces such as sockets and TLI are often very apparent in network bandwidth-intensive applications. [1] provides discussions regarding performance trade-offs related to the use of various transport mechanisms over ethernet and ATM networks and suggests that for certain applications, the use of lower level interfaces may be required for performance reasons. This trade-off, however, can result in a much more expensive development task if performance critical portions of the network interface make up a relatively small part of the overall implementation. Some efficiencies can be gained back through the use of C++ type-safe wrappers for the traditional C style network programming interfaces such as the ACE components[2], but distributed object semantics like those provided by CORBA are not provided by these wrappers.

The patterns described in this paper provide a mechanism that allows use of standard buffering techniques to improve performance of certain distributed objects. The buffered collection and buffered iterator patterns allow efficient handling of collections of small primitive elements while supporting conventional distributed object semantics through remote proxy objects[3][4]. Specifically, the patterns apply to remote proxies for collections that are composed of very large numbers of small primitive elements in collection form.

These patterns are motivated by the following forces:

- latency - networks, RPCs, process context switching all add up to a great deal of latency when a single element is transferred.
- efficient transfer - different methods of data transfer have different limitations and different performance characteristics.

- distributed object semantics - users of distributed objects should be unaware of the distributed nature of the objects they are using.
- simplicity - This pattern can be implemented without committing to the administrative complexities and version and vendor dependencies of distributed object systems like CORBA. At the same time, it can be implemented in conjunction with these systems where efficiency provided by the distributed object system is not adequate by itself.

Using these patterns, data can be efficiently transferred by a combination of

- *specialized buffering* of the data to be transferred.
- *optimized inter-process communication* allow use of higher performance communication techniques like shared memory or sockets.

while maintaining the distributed object semantics of the Remote Proxy pattern.

This paper is organized in the canonical design pattern format described in [3]. Section 4 motivates these patterns by showing how they are used in Mentor Graphics' Distributed Design Rule Checking system ICrules™. Sections 5-8 describe the patterns in detail. Section 9 describes implementation in which a single address space application is converted to a multi process application. Section 10 describes a sample distributed List and ListIterator that use the pattern.

2.0 Also known as

Buffered Container

3.0 Intent

Provide an efficient and flexible mechanism for handling bulk data efficiently across high latency networks, while presenting an interface that is similar to traditional collections and iterators.

4.0 Motivation

To illustrate the Buffered Collection and Buffered Iterator patterns, consider the Distributed Design Rule Checking system ICrules from ICverify, a physical design verification system for integrated circuit design.

4.1 Integrated Circuit Verification

Integrated Circuit (IC) verification design tools are designed to check or "verify" various aspects of an IC design. The IC layout is the geometric information that will eventually be used to manufacture a chip using a specific IC manufacturing process.

Design Rule Checking (DRC) verification performs a set of user defined geometry checks on an IC layout. The purpose is to make sure that a physical design conforms to the constraints of a given IC manufacturing process. For example, a given manufacturing process may require that metal conductor lines are at least 1 micron in width and that they are separated from one another by 2 microns.

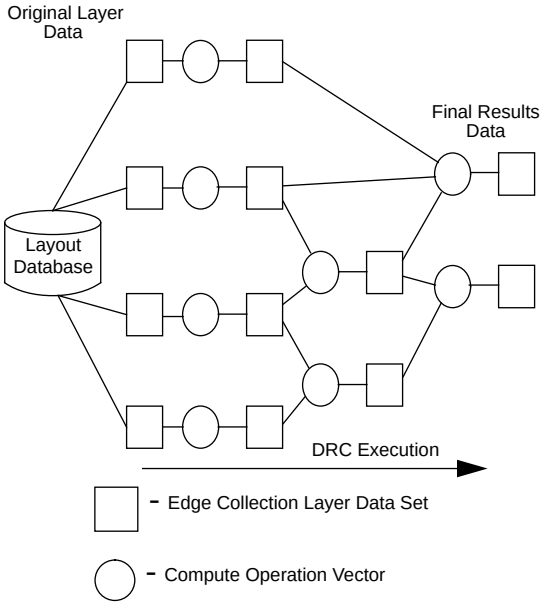
DRC algorithms used in ICrules are characterized by large independent design data collections called Edge Collections and compute intensive operations called Operation Vectors:

- Edge Collections - large data collections composed of simple line segment geometries.

- Operation Vectors - Analysis is performed using the Edge Collections as input. New Edge Collections are created as output.

Figure 1 illustrates the exeution sequence of a typical DRC run. Since any two operation vectors can proceed in parallel without interaction if their respective input data layers are available, the “divide and conquer”[7] strategy to tackling parallel problems is ideal for the DRC algorithms used in ICverify. Tasks can be partitioned and scheduled by a master process and executed by a number of slave processes that are distributed through a network. This coarse grained parallelism is also particularly applicable here since the data transfer time is often small compared to time required for execution of an Operation Vector.

FIGURE 1. DRC Execution



While the cost of transfer for an Edge Collection tends to be small compared to the Operation Vector, the Edge Collections are still often very large. Edge Collection data transfer can still be an expensive part of the overall parallel DRC run. Therefore, efficient manipulation of these collections in a distributed environment is essential to the success of this design.

4.2 Distributed Data Collections

ICrules’ Edge Collection is composed of a very large number of small primitive data elements. The primitive input data element for the Edge Collection is either a single line segment (two x,y coordinate pairs) or a polygon (n x,y coordinate pairs).

In the ICrules single address space architecture, these elements are added individually to a large number of Edge Collections as a layout database is read. All access to the collection’s contents is through a set of iterators. Since the underlying code for the Operation Vectors and the Edge Collections and iterators is directly reused from the single threaded version of the application, the distribution of the operation vectors relies on efficient writing, reading, and copying of the Edge Collections and their contents between the master and the various slave processes.

One way to process these collections in parallel in the distributed system is to add them element-by-element as they are in the single address space version. In general, for systems that are being converted from single address space system to parallel systems, this approach allows the fundamental data handling mechanisms to work much the same way that they did before. Element by element checking or verification can happen in the same way that it would on a single address space solution. A large number of independent collections can be handled simultaneously without extraordinary memory requirements. This approach breaks down, however, when the data sets become very large. In this case, latency for passing individual elements across a network connection is unacceptably high.

Another approach is to create an entire collection in the local address space and then send it en masse to the remote address space whenever it is needed remotely. This addresses the cumulative latency issue since the collection can be transferred much more efficiently as a unit. However, it may place unacceptable memory requirements on the system since multiple complete copies the collection are in existence simultaneously. It can also limit simultaneous handling of large numbers of collections again by placing unacceptable memory requirements on the client. It can also become unacceptably expensive if the collection must be transferred to the remote address space and back repeatedly through the course of execution.

Often, a more efficient means of processing this information is to use the Buffered Collection and Buffered Iterator patterns. *These patterns collaborate to provide an efficient mechanism for handling bulk data efficiently across high latency networks, while presenting an interface that is similar to traditional collections and iterators.*

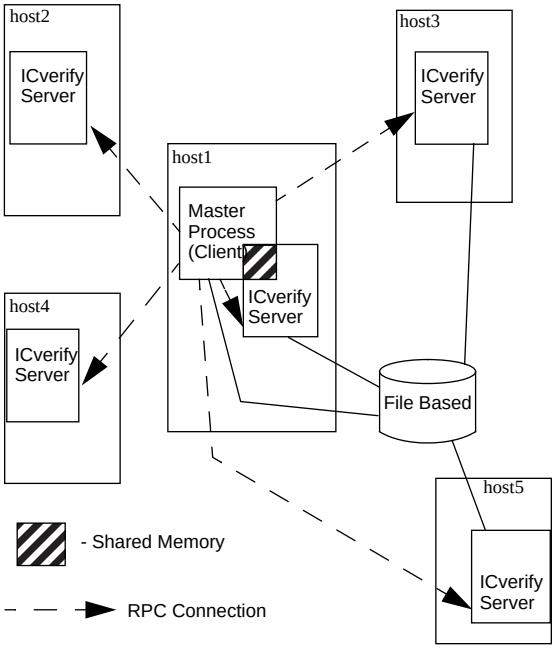
By taking advantage of sequential or predictable patterns of loading and accessing individual data elements in a collection, this pattern encapsulates efficient bulk data transfer among distributed objects. It also encapsulates several forms of highly optimized bulk data transfer.

These patterns provide the following benefits for the ICverify system:

- *Ease of use* - the interface looks like a typical collection and iterator.
- *Efficient buffered data transfer* - this overcomes the issues of latency in interprocess communication.
- *Optimized data transfer techniques* - further efficiencies can be achieved through use of optimized data transfer techniques that are particular to the platforms and locality of the processes involved.

Figure 2 presents a depiction of Distributed ICrules using the buffered collections and iterators. Shared memory buffers are used on the server that resides on the same host as the master process. RPC buffers are used to communicate with remote machines. File based collections can be used for data used by multiple servers.

FIGURE 2. Distributed ICrules



5.0 Applicability

Use the buffered collection and iterator pattern when distributed processing applications have the following characteristics:

- Collection data involves a large number of small primitive elements.
- The elements can be added to the collection without immediate interactions or side effects.
- Performance constraints require multiple optimized methods of bulk data transfer.

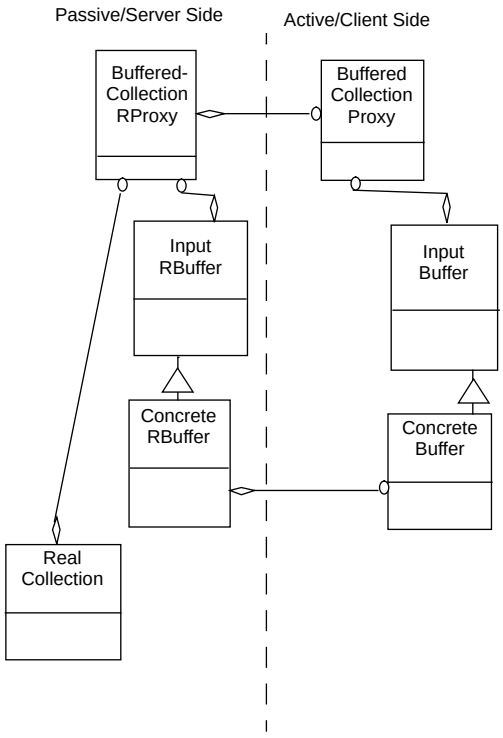
- Simple iteration requirements. The pattern is not applicable when complex caching is required. It is also not suitable for “robust iterators” or iterators that can cope with insertion and removal won’t interfere with traversal[3].

6.0 Structure and Participants

The participants in each of these patterns act in client and server pairs. The server side of the pair is distinguished by the “R” in its name (for Remote).

6.1 Buffered Collection

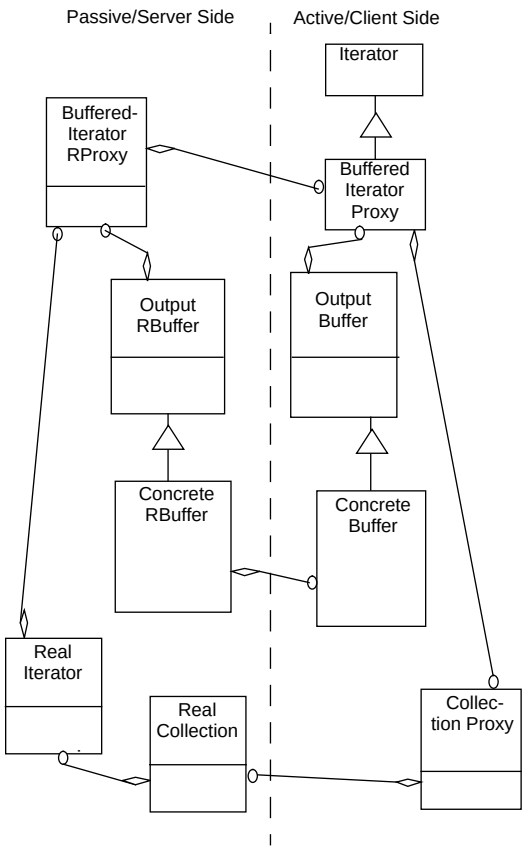
FIGURE 3. Buffered Collection



- Buffered Collection proxy - A client application uses this interface as if it were the actual collection. Some user-visible methods may be required for flushing of buffers or to indicate end-of-data. Often these can be hidden within the other methods of the proxy.
- Buffered Collection RProxy - The remote side of the object proxy. This contains the real collection.
- Input Buffer/RBuffer - Buffered Collection uses this abstract interface to buffer up data for transmittal data to the actual collection.
- Concrete Input Buffer/RBuffer - Implements a specific type of Input Buffer/RBuffer. This controls method of data and control transfer, size of the buffer, flushing and synchronization. These classes are derived from Input Buffer/Rbuffer. These are added to the proxy classes via a factory pattern.
- Real Collection - Reuse a non-distributed Collection.

6.2 Buffered Iterator

FIGURE 4. Buffered Iterator



- Buffered Iterator proxy - This proxy is derived from an Iterator as described in [3].
- Collection Proxy - The Buffered Iterator proxy “knows about” a Real Collection through a Collection Proxy that need not be a Buffered Collection Proxy.
- Output Buffer/RBuffer - Buffered Iterator proxies use the Output Buffer interface to pull data from the actual collection.
- Concrete Output Buffer/RBuffer - Similar to the Collection’s Input Buffer/ RBuffer. Derived from Output Buffer/ RBuffer.
- Real Iterator - Reuse a non-distributed Iterator.

Two specific types of Concrete Input Buffer/RBuffer are implemented in the example. The two in the example are:

- RPC Buffer/RBuffer - uses ONC RPC as the underlying data transport. This provides a very flexible transport mechanism useful under general circumstances.
- Shared_Malloc Buffer/RBuffer - uses ACE Shared_Malloc[2] as the underlying data transport. This provides a transport mechanism that is optimized for same-host execution.

Other transport mechanisms could be encapsulated here as well. Two additional transport mechanisms that might be useful are:

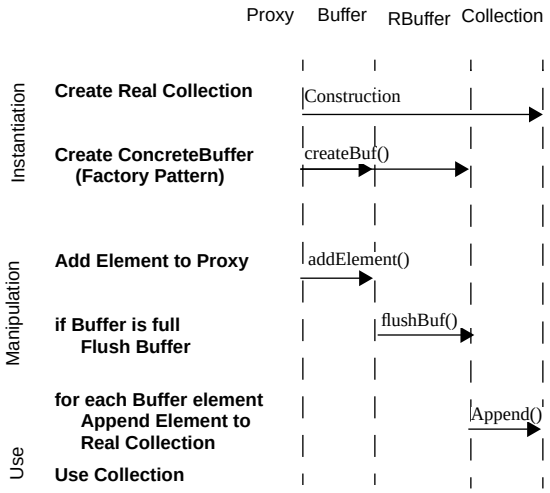
- File Buffer/RBuffer - uses NFS mounted files as the underlying data transport. This can provide an especially efficient mechanism for broadcasting data to multiple remote address spaces since the single file can be read by multiple remote processes.
- Sock SAP Buffer/RBuffer - uses ACE socket wrappers[2] to provide an efficient mechanism for transfer where hosts are known use consistent data representations. This mechanism removes much of the RPC marshalling and unmarshalling overhead associated with the RPC Buffer/RBuffer. Performance gains related to the use of this type of transport mechanism are discussed in detail in [1].

In addition, data compression techniques can be applied to the buffered data to further reduce network transmission overhead. For example, ICrules uses a number of compression techniques including representing orthogonal rectangles (usually four x,y coordinate pairs) with two x,y coordinate pairs representing the opposite corners of the rectangle.

7.0 Collaborations

The collaborations in the buffered collection and buffered iterator patterns are illustrated by the following object interaction diagrams.

FIGURE 5. Buffered Collection Interactions



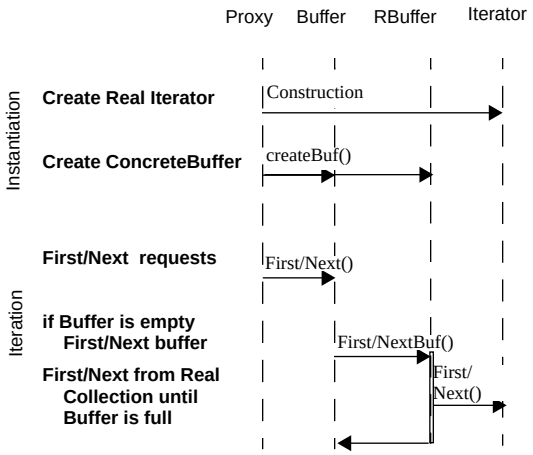
7.1 Buffered Collection

1) *Instantiation.* Construct the proxy without a buffer. The proxy will take care of constructing the remote collection. The buffer can be created using a factory pattern to avoid hard-coding the underlying transport mechanism into the collection proxy.

2) *Manipulation*. Data may be added either through the remote proxy or directly to the collection but not through both channels. Buffers flush themselves as required. Other methods explicitly flush the buffer when they need to make sure that the Real Collection contains all current data.

3) *Use.* Use the collection either directly at the server or through the Buffered Iterator pattern.

FIGURE 6. Buffered Iterator Interactions



7.2 Buffered Iterator

1) *Instantiation.* Construct the Buffered Iterator proxy on an existing Collection Proxy without a buffer. The iterator proxy will take core of constructing the remote iterator. The buffer can be created using a factory pattern similar to that for the Buffered Collection.

2) *Iteration.* Read data from collection via the Iterator Proxy. The buffer will pull data as it is required.

8.0 Consequences

There are several benefits to using the Buffered Collection and Buffered Iterator patterns:

Encapsulation of optimized data transfer techniques. These techniques include high-speed networks or lower latency data exchange mechanisms (such as shared memory through a facility like ACE Shared_Malloc [2], sockets, or disk file sharing). Encapsulating the optimizations behind a consistent abstract interface simplifies implementation by allowing one consistent general purpose mechanism (such as RPC) to handle the bulk of the implementation. This allows improved

primitive element transfer performance through alternate methods like shared memory without redesigning the bulk of the distributed implementation.

Ease of Use. The buffered collection allows a collection object to be filled one element at a time while avoiding the overhead of transferring elements one at a time. Careful construction of the proxy can completely hide the buffer when all access to the client occurs through the Buffered Collection Proxy.

Improved portability. Portability is improved through encapsulation of efficient data transfer methods.

Simplicity of distributed object mechanism. Use of this pattern does not require large scale and potentially difficult to administer distributed object implementation frameworks like CORBA, although it is not precluded when they are in use.

There are also several drawbacks to using this pattern:

Input Restrictions. Input of data elements must occur without regard to element by element state changes in the collection or other dependent entities.

Input checking or verification restrictions. Checking of individual data elements has one of the following characteristics:

- verification is not required.
- verification can occur independently of the bulk of the collection.

Complex caching restrictions. These patterns have minimal cache consistency checks and are therefore limited to fairly simple usage. Many of the issues described in relation to the buffered collection and iterator apply to a more complex distributed data implementations. In addition, the more complex forms require something like an observer pattern[3] to guarantee cache consistency.

Examples of services offering more complex distributed data management include the DCE distributed file system[10] from OSF, and Linda[11] distributed shared memory system. OSF's DCE implements a token passing scheme to guarantee consistency between multiple cached copies of pages from distributed files that are a part of its distributed file system. Linda provides a virtual shared memory system which also guarantees the consistency of multiple cached copies of data.

More complex data distribution mechanisms like these are required for the following situations:

- Robust Iterator pattern[3] - Gamma et al describe a pattern called the Robust Iterator in which a collection can be altered while an iterator is running and the iterator will behave in an expected manner (i.e. it should not return items that have been previously deleted from the collection). The simple buffering mechanisms in this pattern rule out such robust behavior.
- Multi Client input - multiple clients can alter data in a single collection and the precise order of this input is important. Again, the simple buffering techniques do not allow for a great deal of communication between the clients and the server, thus ruling out this type of application.
- Non-iterative retrieval of the elements of the collection.

9.0 Implementation

The Buffered Collection and Buffered Iterator patterns are particularly useful for adding parallel execution capabilities to existing single process/single thread appli-

cations because of their similarity to normal collections and iterators. This section describes steps that might be taken in implementing this pattern while converting such a single process architecture to a multi-process architecture.

9.1 Initial Partitioning

Before considering the buffered collection and iterator, it is necessary to construct a viable multi-process architecture and to partition the objects and algorithms used across the participating processes.

9.2 Collection/Iterator proxies

Once the objects are partitioned, some objects that require representation across the client/server boundary will require remote proxy object representation. Others, like the Operation Vectors in ICrules will be simple and small enough to be copied from the client to the server and back when they need to be accessed. At the early stages, these remote proxy collections can be designed as unbuffered proxies, thus simplifying initial implementation and allowing additional analysis of proxy performance and buffering requirements.

The unbuffered mechanisms can also often be used as a fall back for infrequent cases. For example, if an element is extraordinarily large, it may exceed the size of a shared memory buffer. Rather than going through the trouble of increasing the size of the buffer for what may be an infrequently encountered object, it may suffice to use a single unbuffered transfer mechanism for very large objects.

Care should be taken in this stage to design collection and iteration interfaces with the limitations described in the Consequences section above in mind.

9.3 Client/Server transport

Next, select transport and control mechanism for the proxies. Since the buffers can support different specialized transport and control mechanisms for performance sensitive aspects of the design, a general transport mechanism can be selected for applicability to the rest of the implementation. For ICrules, ONC RPC provided a combination of portability and ease of administration on existing Unix networks that made it particularly appealing. DCE or CORBA could also be used to provide general typed communication and control across the process boundaries. Similarly, more primitive or restricted environments can be used to implement this pattern as well.

9.4 Buffer Requirements

Once the proxies are in place, the buffering mechanisms can be implemented and optimized with alternate transport mechanisms if necessary. The design of the buffer itself will need to take into account the following:

Sizing strategies - These are often dictated by the transport mechanism being used. For memory-mapped files, file size restrictions may come into play. For RPC mechanisms, network buffer sizes may dictate ideal buffer sizes.

Transport Facility Drawbacks - The transport facilities discussed all have some drawbacks to consider in their selection.

- *ONC RPC* - provides the most general transport mechanism of those discussed here. It does so with some additional expense required for marshalling and unmarshalling of data required for generalized transport.

- *mmap shared memory* - provides high performance same-host transfer, but is limited to same-host execution. Since data is shared through the file system, file cleanup and disk management issues also exist.
- *System V shared memory* - The interface for system V shared memory also provides high performance same-host transfer. It has associated limits on memory segment size, segment cleanup.
- *sockets* - provides a middle ground between the generality of RPC connections and the performance of shared memory. It is usable across machines with similar data representations.

Used in combination, these transport facilities can provide a combination of high performance, portability, and generality to the application.

Control and Synchronization - Control issues also need to be dealt with for each buffer implementation:

- *ONC RPC* systems provide their own control and synchronization mechanism that provides a simple mechanism in which the client blocks until the server returns. More elaborate message style control mechanisms can be constructed[5] to enable parallel processing in the client and the server.
- *mmap based* - shared memory does not, by itself, provide any mechanism for synchronization, so some other mechanism, such as semaphores, must be selected. If the proxies are constructed using RPC, a simple RPC based control mechanism is easy to implement.
- *file based* - record locking [9] can provide an efficient and simple mechanism for file based control, although NFS locking capabilities are not well known for their robustness.
- *socket based* - Mechanisms here can be similar to those used by RPC systems themselves. For example, record structures etc. may provide an efficient mechanism for determining when the end of data has been reached.

Compression - finally, compression techniques may be applied inside the buffer especially for the lower bandwidth transport mechanisms like RPC or sockets.

10.0 Sample Code

The following section illustrates a simple List class and List iterator that are implemented as a BufferedListProxy and BufferedListIteratorProxy. Two buffer implementations are illustrated, the first uses RPC and the second uses Shared_Malloc_MM from the ACE class library[2]. For simplicity and brevity's sake, most of the error handling code and C++ const correctness has been omitted.

The List class is a shortened version of the List class described in [3]. Many of the member functions of the class have also been removed for brevity's sake.

```
template<class Item>
class List
{
friend class ListIterator<Item>;
public:
   List();
   ~List();
   unsigned int Count();
   bool Prepend( Item & d );
   bool Append( Item & d );
   bool RemoveFirst( Item & d );
};
```

The ListIterator presents the Iterator interface described in [3]:

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() = 0;
    virtual Item CurrentItem() = 0;
protected:
```

```
    Iterator();
};
```

The BufferedListProxy class serves as a wrapper that provides all of the List class interface:

```
template<class Item>
class BufferedListProxy :
   private DistributedObject
   //DistributedObject
   //provides multi-process object
   //synchronization for proxy objects
{
// friends ...
public:
   BufferedListProxy(int clnt_id);
   ~BufferedListProxy();
   unsigned int Count();
   void Prepend( Item & d );
   void Append( Item & d );
   bool RemoveFirst( Item & d );

private:
   ItemBuffer<Item> *itemBuf;
};
```

The ItemBuffer class provides an abstract interface to the Buffer:

```
template<class Item>
class ItemBuffer
{
public:
   virtual void addElement( Item& ) = 0;
   virtual void flushBuf( ) = 0;
};
```

A Concrete ItemBuffer for RPC's is declared as:

```
template<class Item>
class RPCItemBuffer : public
ItemBuffer<Item>
{
public:
   RPCItemBuffer(
     BufferedListProxy<Item>&,
     unsigned long NElements = 32
   );
   virtual void addElement( Item& );
   virtual void flushBuf( );
private:
   class BufferedListProxy<Item>& momma_;
   unsigned long n_elements_;
   void* buffer_;
};
```

The BufferedListProxy in the example uses Polygons that are a list of two point vertices in ONC rpcgen .X declaration form:

```
struct vertex {
    int   x;
    int   y;
};

struct polygon {
    /* variable length array of vertices */
    vertex     vertex_info<>;
};
```

Mixed use of the OO features of C++ and a strictly 'C' interface like ONC RPC leads to the need for providing some transition from objects to pure C datatypes and back again while avoiding excessive copying of data structures while they are decomposed in preparation for transmittal. A C++ class Polygon is derived from the "C" style polygon class in order to provide some expected object semantics. It is the primary object. Private member functions that accept the c style polygon as a parameter are provided to allow direct creation of Polygon objects from the transmitted RPC data.

```
class Polygon : public polygon {
friend ... ;
private:
   Polygon(polygon&);
   Polygon& operator=( polygon& );
public:
   Polygon();
   Polygon(Polygon&);
   ~Polygon();
   Polygon& operator=( Polygon& );
};
```

The BufferedListProxy<Polygon> can now be declared and manipulated.

```
// Create a list on remote process clnt_id
BufferedListProxy<Polygon> A_poly_list(
   clnt_id
);

// add an RPC based buffer
AddItemBuffer( A_poly_list, RPC_BUFFER );
```

Most methods for the proxy are implemented by breaking down the objects involved into C style structs and passing them via RPC. Direct interaction with the RPC functions requires that the proxy classes be explicitly instantiated. The

Append() method serves as a good example here. Other methods are similarly implemented. Append first checks for an Item Buffer and uses addElement(Item&) if it can. If it can not, it packages the polygon into a form ready for transmittal via RPC and uses the rpcgen generated polylist_append_1() to transmit the polygon.

```
void
BufferedListProxy<Polygon>::Append(
    Polygon& p
)
// Insert a Polygon at the back of the list
{
   if ( itemBuf ) {
      // Add element to the item Buffer.
      itemBuf->addElement( p );
   } else {
     bool *b_rtn;
     polygon_info poly;
     poly.id = remote_id;
     // Assign to the RPCGEN generated
     // vertex<> structure
     poly.p_val.vertex_info.vertex_info_len
        = p.vertex_info.vertex_info_len;
     poly.p_val.vertex_info.vertex_info_val
        = p.vertex_info.vertex_info_val;
      b_rtn
        = (bool*) polylist_append_1(
                &poly,
                client_hndl->rpc_client()
              );
     }
}
```

In the case where the itemBuf exists the ItemBuf<Polygon>::addElement(Item&) is then responsible for adding the element to the buffer. A polygon_block is an RPC structure for a variable length list of polygons. This structure is used by the RPC buffer to transmit a block of polygons from the buffer.

```
void RPCItemBuffer<Polygon>::addElement(
       Polygon &P
      )
{
   polygon_block* pb =
      (polygon_block*) buffer_;
   if ( pb->poly_info.poly_info_len >=
           n_elements_ ) flushBuf();
   assign_from_poly (
       pb->poly_info.poly_info_val[pb->
          poly_info.poly_info_len++],
       P );
}
```

The flushBuf() method sends the cached polygons in a polygon_block. Other methods like RemoveFirst() may call flushBuf() before doing the removal or, for greater efficiency, try to figure out if the first element in the list is in the buffer and then manipulate the buffer directly.

```
void RPCItemBuffer<Polygon>::flushBuf()
{
   polygon_block* pb =
        (polygon_block*) buffer_;
   if (pb->poly_info.poly_info_len != 0 ) {
      polylist_append_rpc_block_1(pb,
         momma_.client_hndl->rpc_client()
      );
      pb->poly_info.poly_info_len = 0;
   }
}
```

The BufferedListRProxy's buffer takes the transmitted items and inserts them into the List. The Iterator Proxy works in about the same way, except the buffers are 'pulled' from the remote proxy rather than being pushed as they are in the List.

The Shared_Malloc_MM buffer is implemented similarly to the RPC buffer except that the buffer is a shared memory buffer. Control is still transferred via an RPC call, but it contains a minimum amount of data and so overhead related to copying the data through the RPC socket and marshalling/unmarshalling are avoided. The addElement method for the Shared_Malloc_MM buffer simply copies polygons into the buffer in a specific format:

```
void Shared_Malloc_MMItemBuffer<Polygon>::
       addElement(Polygon & P)
{
  // flush the buffer if the cache is full
  if ( buffer_would_be_full( P ))
      flushBuf();
  // Handle the case where the buffer
  // is not big enough for a single
  // element.
  if ( buffer_would_be_full( P ) )
  {
    // use RPC method, that can handle
    // arbitrary size.    (alternatively we
    // could grow the shared memory buffer.)
    bool *b_rtn;
    polygon_info pi;
    pi.id = momma_.remote_id;
```

```
    // assign to the RPCGEN generated struct
    pi.p_val.vertex_info.vertex_info_len
      = P.vertex_info.vertex_info_len;

    pi.p_val.vertex_info.vertex_info_val
      = P.vertex_info.vertex_info_val;

    // Make the RPC call
    b_rtn = (bool*) polylist_append_1(
        &pi,
        momma_.client_hndl->rpc_client()
    );
  } else {
    // writeData is a simple helper that
    // copies into the shared memory buffer
    // via memcpy and updates the cursize_.
    writeData(
      &P.vertex_info.vertex_info_len,
      sizeof(P.vertex_info.vertex_info_len)
    );
    writeData(
       &P.vertex_info.vertex_info_val,
       P.vertex_info.vertex_info_len *
       sizeof( vertex )
    );
   }
}
```

11.0 Examples

Distributed ICrules

Batch RPC on ONC RPC [5]

DCE Pipes [8]

Corba Event Channels

12.0 Variants

Variations or enhancements to this pattern might include:

- Use of cache consistency algorithms to allow for a more complex collection/ iterator interactions like the Robust Iterator.
- Support for multi-client manipulation of a server side of the collection.
- Page based retrieval systems that might allow efficient retrieval of elements the collection in situations that aren't strictly iterative.
- Lazy Evaluation of the buffered data - some systems might benefit from collections which transmit and marshall/ unmarshall buffered data on demand rather than as it is added to the collection.

13.0 See Also

Remote Proxy pattern [3], [4]

Iterator pattern [3]

14.0 Acknowledgments

The following individuals contributed to the development and refinement of this pattern:

Douglas Schmidt - schmidt@cs.wustl.edu

Rich Strobel - rich_strobel@mentorg.com

Robert Todd - robert_todd@mentorg.com

George. Moberly - georgem@homer.atria.com

Participants in the OOPSLA 1995 workshop on Concurrent, Parallel and Distributed systems.

15.0 References

[1] Douglas C. Schmidt, Tim Harrison, Ehab Al-Shaer, "Object-Oriented Components for High-Speed Network Programming", Proceedings of the USENIX Conference on Object-Oriented Technologies, Monterey, CA, June 1995.

[2] Douglas C. Schmidt, "The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/ Server Applications", Proceedings of the 12th Annual Sun Users Group Conference, SUG, San Francisco, CA, pp. 214-225, June, 1994.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Reading, MA, 1994

[4] A. Birrell, G. Nelson, S. Owicki, E. Wobber, "Network Objects," Digital Systems Research Center, Feb 1994

[5] John Bloomer, "Power Programming with RPC," O'Reilly & Associates, 1992.

[6] G. Hamilton, M. L. Powell, J. G. Mitchell, “Subcontract: A Flexible Base for Distributed Programming.” Proc. 14th ACM Symposium on Operating Systems Principles, December 1993.

[7] E.V. Krishnamurthy, “Parallel processing: principles and practice,” Addison-Wesley, 1989.

[8] J. Shirley, W. Hu, D. Magid “Guide to writing DCE Applications,” O’Reilly & Associates, 1992.

[9] W.R. Stevens “Advanced Programming in the UNIX Environment,” Addison-Wesley, 1992.

[10] Open Software Foundation, “DCE Filesystem white paper”, http://www.ofs.org/comm/lit/OSF-O-WP8-0990-2.ps

[11] N. Carriero, D. Gelernter, “Linda in Context”, Communications of the ACM Vol32 April 1989.