NTCluster DataPump, Rivers, and Sorting

Joshua Coates, Joe Barrera, Alessandro Forin, Jim Gray,
Microsoft Bay Area Research Center
jcoates@CS.Berkeley.edu, { JoeBar, SandroF, Gray} @Microsoft.com
August, 1998

Abstract

We report on the design, implementation, and performance of three distributed systems running on a cluster of WindowsNT nodes: DataPump, RiverSystem and NTClusterSort. The *DataPump* is a simple data transfer program that has twice the throughput and 75% of the CPU cost of the default copy methods. The *RiverSystem* redistributes data across a cluster using a collection of point-to-point data streams. *NTClusterSort* is a sample application based on the RiverSystem that performs distributed one pass and partial two pass sorts.

1. Introduction

It is common to find large-scale clusters of commodity PC's or workstations with local storage linked together by a system-area network. They are typically found in web-farms, mail servers, and in scientific applications. Leveraging commodity components to achieve inexpensive scalability, parallelism and fault tolerance is the key to making these systems worth the trouble. However, it is still difficult to program and manage clusters having hundreds of processors, gigabytes of memory, and terabytes of data spread across the cluster. Tools built for single nodes are typically useless when applied to distributed systems. This paper describes a method for structuring dataflows and computations within a cluster using a combination of DCOM and streams.

The core idea is that data records can be treated as a fluid that flows from *data sources* to *data sinks*. Each node (source or sink) of the flow has one or more sequential input record streams and one or more sequential output record streams. The sources and sinks can be cloned to get *partition parallelism*. The sources and sinks can be concatenated to get *pipeline parallelism*. The combined flows form a homogenous set of sources to a homogenous set of sinks is called a *data river.* [Barclay]

We developed this system with the source code made available to the public in hopes that it can be configured, adapted and improved for a multitude of distributed applications. The source code is available at http://research.microsoft.com/barc/ .

This paper describes the system in three sections. We first describe the fundamental building block of distributed data management, the *DataPump* that moves data from a single source to a single sink. We describe the concept, the implementation and the performance of the DataPump, comparing it to conventional method of moving data in a cluster. Next, we describe the design and implementation of the RiverSystem , which moves data from multiple homogeneous sources to multiple sinks. Each source or sink deals with a put-record or get-record abstraction. We discuss the DataRiver's performance and bottlenecks. In the next describes NTClusterSort, which is a simple application of the RiverSystem. NTClusterSort implements both a one-pass and a partial two-pass sort (the merge phase is incomplete.) Then the performance of NTClusterSort's is discussed. The paper concludes with a brief discussion of how the DataPump, RiverSystem, and NTClusterSort could be installed and launched using DCOM as the installation and execution engine. A simple GUI has been built that installs the application and benchmarks it on an NTCluster.

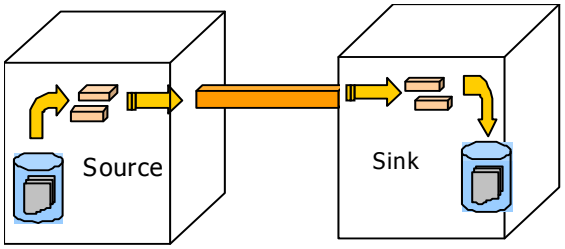

Figure 1. The DataPump overlaps reads from disk and writes to the socket on the source node. On the sink node, it reads from the socket and overlaps write to the disk. Data is stored in buffers between disk and sockets.

2. DataPump

Transferring data from point A to B is the first step in

1

managing large data sets. It is important to first understand the issues involved in this basic process.

The DataPump moves data from a source to a sink. It is the basis for the more complex distributed applications of NTRiverSystem and NTClusterSort.

The design for the Data Pump is essentially John Vert's Unbuffered Copy program from the Win32 SDK. The difference is that the source sends its data to a Windows socket [Winsock version 2.0] rather than a remote file. The sink reads from a socket rather than reading from a file. WindowsNT supports a common API to both files and to sockets. So the source and sink programs are essentially identical -- and they are essentially identical to Verts' file-to-file copy program. The source process reads data from a disk, and pumps it through a socket to the sink process. The sink process reads the data from the socket and writes it to disk.

```
while(PendingIO) {
   WaitForIOCompletion();
      If (Read_Completed)
        WriteFile();
      If (Write_Completed)
        if(EOF)
          PendingIO--;
          Continue;
        else
          ReadFile();
}
```

Figure 2. Simple algorithm for the Data Pump. Both the source and sink processes execute the same program. Note that ReadFile() and WriteFile() may refer to disk or socket I/O.

The inner loop of DataPump source and sink is shown in Figure 2. ReadFile() and WriteFile() take a Handle as an argument which can either refer to a file or a socket. The reason both the source and sink processes can share this algorithm is because in a sense they are doing exactly the same thing, reading from a source, and writing to a sink.

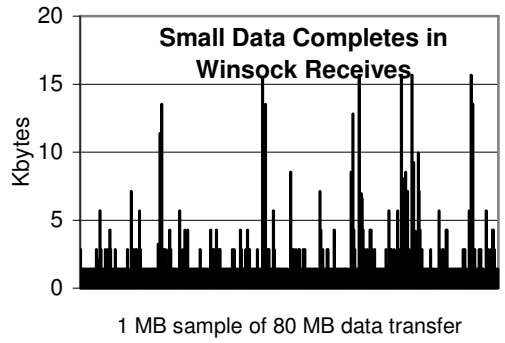

Figure 3. Requesting 64K on a receiving socket typically returned less than 5K of data. To fill a 64K buffer, recv() needs to be called 40 times on average. This contributes to the higher CPU rate on the destination node. As the graph indicates, 16K completions were rare.

The difference between reading a file and reading a socket is that socket reads do not generally return all the requested data at once. When a 64KB disk read request is made, all the requested data is returned immediately, unless there are errors or the request crosses the end of file. A 64KB socket read request returns whatever data was available at the time of the call. We found that requesting 64KB of data on a socket typically returned 2KB worth of data at a time. (See figure 3.) This creates to a serious performance problem discussed in the next section. Because of this Winsock nagling[1], when a Winsock read completes, the sink checks the amount of data received, and if it was not the full amount, another asynchronous read is issued for the remaining data. This complication is not included in Figure 2, but would be handled under the Read_Completed clause.

Performance

The performance experiments were performed on 333MHz Pentium II machines running NT4.0 SP3. The disks typically could be read and written at 9 MB/s with Write Cache Enabled [Riedel].

CPU:	Pentium II 333 MHz
Memory:	256 MB DRAM
Disk:	9 GB SCSI
Network:	100 Mbps Ethernet

Figure 4. The performance experiments where conducted on machines with these properties.

As a reference to DataPump's performance, we ran tests using the built in "Copy.exe" program as well as "RoboCopy.exe" from NT Resource Kit. We copied files from a local drive and let Copy.exe and RoboCopy.exe use

2

the NT redirector service, which is NT's network files system to access the remote destination. Both "Copy.exe" and "RoboCopy.exe" had similar throughput and CPU usage: ~4.5 MB/s, throughput, 35% cpu utilization on the source machine, and about 45% utilization on the sink.

The DataPump's throughput, 9MBps, was approximately twice as much as the default copy programs. We attribute this to overlapping the reads and writes to the disk. As disk reads complete, new disk reads are issued. Always having an outstanding read request assures that the disk is always busy, and the processor is free to send data to the

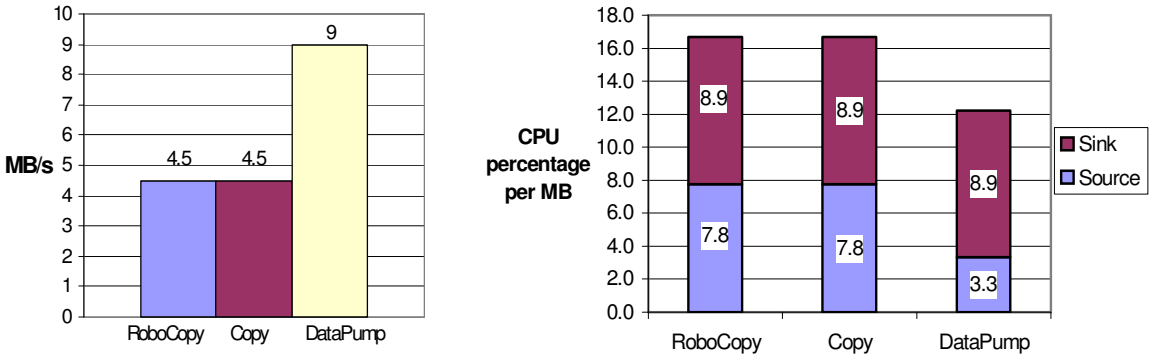

Figure 5: The DataPump's throughput is twice that of Copy.exe or RoboCopy.exe, but uses approximately 25% less CPU per MB transferred. This is because DataPump consumes proportionally less than have the CPU on the source node. The destination cpu consumption is comparable for all three programs, reflecting the Winsock nagling problem.

sink. On the sink node, as data arrives, they are immediately outstanding written to disk. This balance of two active disks and two active processors working on data transfer gives a twofold throughput increase.

The DataPump's source CPU utilization is approximately 25%, or about 80 million CPU clocks per second on a 333 MHz machine. Since the source pumps 9MB/s, this is approximately 9 clock cycles per byte (cpb) (80 Mclocks/9 MBps). The CPU usage is almost 2.5 times greater at the sink: 80-90%, which by the same approximation works out to be 30 cpb.

We believe the reason for this disparity between source and destination is twofold. First, the current implementation of Winsock makes a copy when data is read on a socket, whereas no copy is made when sending. While the cost of this additional copy is significant, it cannot account for the large CPU disparity (30 cpb vs. 9 cpb). The second factor is that the average size of a socket read completion was only 1.6 KB (see Figure 3). Hence, ReadFile() and GetQueuedCompletionStatus() must be dispatched about 40 times per buffer on the sink. We believe that the 3x CPU penalty lies in this 40-fold increase in read dispatches due to Winsock nagling. There are several sockopts which we attempted to use to alleviate this problem (TCP_NODELAY, SO_SNDBUF, SO_RECVBUF) but they seem to be ignored. When these sockopts are implemented, it should reduce the CPU cost of the copy from 39 cpb to less than 20 cpb. As an aside, we also set the Registry key TCPWindowSize, in hopes that it would improve performance, but it had no noticeable effect.

[1] On the send side, the Nagle algorithm enables sending data to be buffered and sent in large chunks in order to minimize packet fragmentation. We use the term 'nagling' to refer to Winsocks current inability to buffer data on the *receive* side before notification of a receive.

The DataPump has a "no disk" option that can be used to measure TCP performance. The no disk option simply disables file reads and writes, and transfers the requested amount of buffers through the TCP stack. We used this option to run experiments on a dual boot machine running NT4.0 and NT5.0 Beta 3. Since the source and destination are on the same node, it is not surprising that the CPU utilization was 100% during a 200MB data transfer. This CPU time was almost completely within the kernel. The NT4.0 boot consistently operates the DataPump at 10.7 MB/s, or 30 cpb. Since running the DataPump with disks on two nodes requires a total of 39 cpb (9 cpb on the source and 30 cpb on the destination) we see that 30 cpb is less than optimal performance due to the CPU bottleneck. The same machine running NT5.0 consistently produces a DataPump transfer rate of 12.5 MB/s, which is 25 cpb. This amounts to an approximate 15% performance improvement over the NT4.0 the TCP stack. We observed that the recv() call on NT4.0 averaged less than 2 KB, but on NT5.0 it averaged 7 KB.

3. I/O River System

The I/O RiverSystem transfers data records from multiple sources across the network to multiple sinks. Conceptually, the "river" is made up from multiple data streams made up of records which "flow" to the correct destination based on some partitioning function. That is to say, if a particular node generates red, green and blue records, the red sink node will receive the red records, the blue sink will get the blue ones, and the green sink will get the greens. The river system manages the partitioning and transport of the records. Source node pumps the records into the river system, and the records flow to their proper destinations, and the sink nodes read the records (see figure 6.)

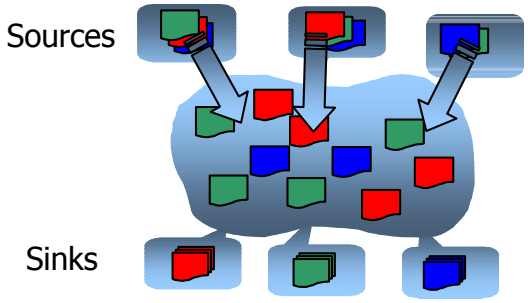

Figure 6. An I/O River System is made up of record sources and record sinks. The records are pumped into the river from the sources and flow to the appropriate sinks. Each sink sequentially pulls records from its input stream.

Design

The semantics of the river system for a node are: PutRecord() for the source and GetRecord() for the sink. Each source sequentially sends a record into the river system, and each sink retrieves a record from the river system. The particular node that makes the request does not have to manage where the record is coming from or where it is going, allowing the application to focus on processing the records rather than dealing with complicated I/O procedures.

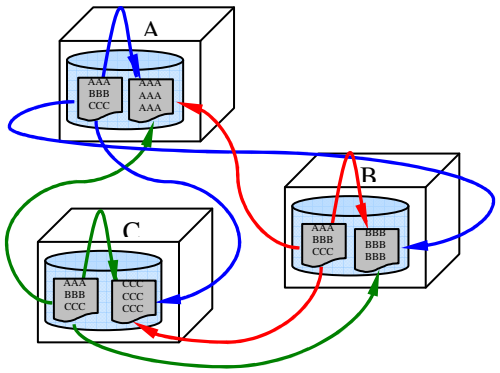

Figure 7. An example of a three node range partition RiverSystem. Node A sends it's B records to node B, C records to node C and A records to itself. Node B and C do likewise.

Internally, the simplest component of the RiverSystem is functionally much like the DataPump. It reads data from disk and pumps data through a socket and reads data from a socket and writes it to disk. However, the RiverSystem pumps to multiple sockets, or "streams" in order to contribute to the data flow of the system wide river. For instance, a range partition is a common data transformation that could be performed by the RiverSystem. A range partition separates records into partitions based on where each record lies within a given range of values. If the RiverSystem is used to perform a range partition on a set of data, the application reads records from disk and uses PutRecord() to send the record to the appropriate socket to its particular partition based on the range criterion. (See figure 7.) At the same time, the node would also be receiving records that

4

correspond with its partition. It would be reading records from multiple sockets and writing them disk.

The RiverSystem initialization requires certain key pieces of information, namely what the hostnames and port numbers are of the participating nodes in the river and also source and destination files for the records. Once this initialization phase has been completed, the river can begin to receive data from a PutRecord() call and distribute data to a GetRecord() call.
An important function which is "river specific" is PickStream(). A RiverSystem used for sorting numbers would use a different PickStream() function than a RiverSystem used for data replicating. This function determines which stream a particular record should be sent through. How this function determines which stream to send a record to is independent to the RiverSystem itself. PickStream() could function as a static hash like a range partition based on a key, or it could just as well be dynamic, picking whichever stream is consuming records the fastest to deal with a heterogeneous cluster. The type of PickStream() function should depend on the type of records as well as the type of application the RiverSystem is intended to support.

On the receiving end, a node in the RiverSystem merely requests records out of the river, and the correct records *flows* to the destination process. The correctness of the record is determined by PickStream() in the source node which sent it.

After PickStream() has determined which the socket the record should go to, the actual record is memcpy'd from the application to a bucket associated with a particular socket. (See figure 6.) This process of partitioning continues until a bucket is full, at which time it is written to the socket. If the data source (the disk) has been exhausted, the remaining partially full buckets are sent followed by an EOF to each socket.

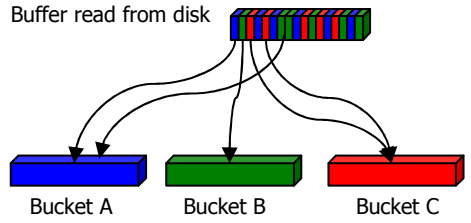

Figure 8. Partitioning the Records in the RiverSystem consists of repeated calls to memcpy() into buckets associated with outgoing streams to the system-wide river. A, B and C represent different types of records an application may want to partition on.

Performance

In our test systems, the CPU is the bottleneck. The majority of the CPU usage is devoted to processing TCP Winsock sends and receives. Unlike the DataPump which either sends or receives data through a single socket, the RiverSystem must handle simultaneous sends and receives through multiple sockets which tends to saturate the CPUs in system.

The experiment consists of starting a RiverSystem process on each of two nodes. A range partition is performed on a set of 2 million records distributed between the two nodes (~190 MB.) The RiverSystem distributes the records to the appropriate node based on the key of each record.

The two-node RiverSystem is able to distribute data at a rate of 7.3 MB/s, or at approximately 44 cycles per byte for our 333Mhz machines. Considering the disk rate is approximately 9 MB/s we conclude that the extra cycles are devoted to the processing of the network calls. It is interesting to note that partitioning of the data is relatively inexpensive when compared to the processing the network calls. Partitioning consists of repeated calls to memcpy() for each record retrieved from disk. This results in over 76,500 sequential calls to memcpy per second, without the benefit leveraging the cache, yet it only accounts for a small (we estimate less than 15%) of the CPU usage in the system.

A similar experiment was conducted on the two-node RiverSystem, but without disks. The disks reads were replaced by reads from buffers preallocated in memory containing records. This experiment allowed us to simulate a system with fast disks and determine the bottleneck. The results were essentially identical to the above results (~7.5 MB/s or 42 cpb), verifying the CPU bottleneck.

4. NTClusterSort

5

The NTClusterSort application illustrates the use of the RiverSystem for distributed applications. NTClusterSort sorts records based on the Datamation criterion of 100 byte records with 10 byte keys. The application was created by simple alterations to both the sending and receiving portion of the base RiverSystem. On the sending side, the PickStream() function was altered to partition the records based on key values. On the receiving side additional threads were added to handle the one-pass and two-pass sorting algorithm. Aside from the one alteration and addition, the RiverSystem was easily transformed into a useful distributed application.

NTClusterSort functions as a one-pass and two-pass sort, depending on available physical memory on each node in the system. The one-pass and two-pass sorts share the initial phase of combining records and sorting them, either into sorted runs or the completed set of records. We will discuss the initial phase common to both, and then discuss the merging phase in two-sort.

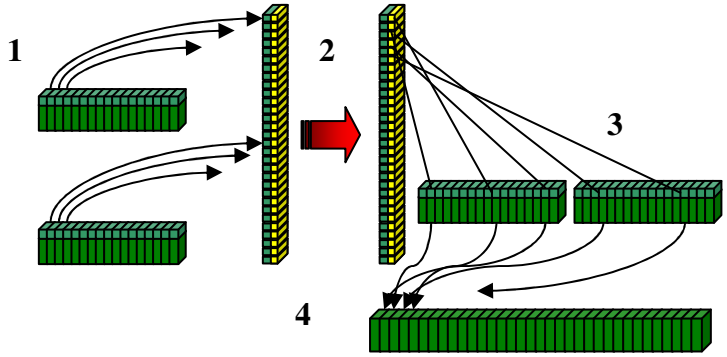

Figure 9. Sorting the records. (1) Keys and record pointers are extracted from incoming buffers into the KeyRecordPointerVector. (2) The vector is then sorted on the keys. (3) The keys are now in sorted order, but the associated records are scattered. (4) Finally, the each record is copied in sorted order to a file buffer to be written to disk as a sorted run.

As records stream in through the RiverSystem, they are passed on to the ExtractAndSortThread, which extracts the keys and pointers to the records and sorts them. The keys and record pointers are copied into a KeyRecordPointerVector, which is either the size of the number of total expected records, or the size of the number of records depending if it is a one-pass or two-pass sort. Eventually the vector is filled and sorted. The internal sorting routine is actually a call to qsort(). (See figure 9.)

The justification for lacking a sophisticated internal sorting algorithm is that we have found through experimentation that the I/O is the dominant cost of sorting and that the sorting algorithm has little affect on overall performance [Gray]. For instance, experimenting with commercially available products, NirtoSort and Postman Sort, which use much more sophisticated algorithms than NT5.0's NTSort we found that despite NTSort's simple use of qsort() for internal sorting, it consistently came within less than 5% of the total average elapsed times of the other two products. We were actually surprised to find that on occasion NTSort would outperform the other two on occasion. These experiments illustrate the high cost of I/O and de-emphasize the importance of fancy sorting algorithms.

The way that NTClusterSort deals with I/O efficiently is through a three-stage pipeline on the receiving end. In this way, both the disk and CPU are never underutilized allowing the application to maximize its efficiency. The stages are as follows: reading form sockets into buffers, extracting key and record pointers and sorting runs, and writing sorted runs to disk. . (See figure 8.) All three operations can be performed simultaneously, each depending on a different resource. Reading from sockets depends on the network, extracting keys and record pointers and sorting depends on the CPU and writing the run file depends on the disk. These three operations are separated into threads and communicate through events.

6

In order to maintain flow through the pipeline, buffers are separated into a FreeList and an ExtractAndSortQueue. After the memory requirements are determined for a particular sort, the newly allocated buffers are put in the FreeList. As data is read from sockets and buffers are filled, they are added to the ExtractAndSortQueue. As the ExtractAndSortThread calls GetRecord(), the next available record is retrieved from the ExtractAndSortQueue. If

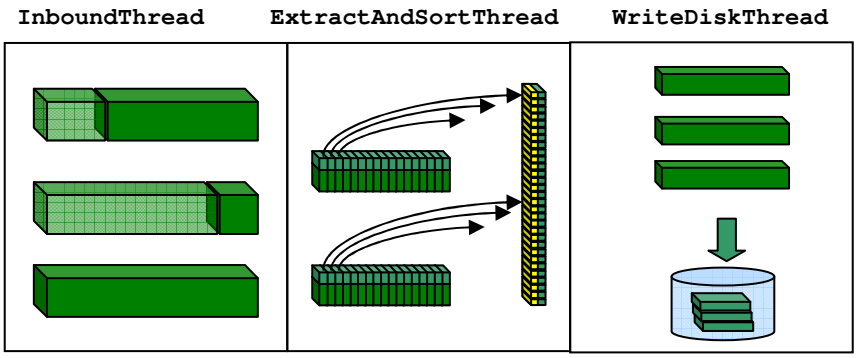

Figure 10. A three stage pipeline is formed in NTClusterSort. ReadSocket() incrementally fills buffers from the sockets. As buffers fill their keys and record pointers are extracted and sorted into runs. As runs are produced, they are written out to disk. These actions can simultaneously utilize the disk and CPU.

no record is available, GetRecord() waits for an event from the queue indicating that a records are available to process. As the ExtractAndSortThread creates sorted KeyRecordPointerVectors, the WriteDiskThread is signaled by an event to copy them to buffers in sorted order and write them to disk.

This process of reading from sockets, creating and sorting KeyRecordPointerVectors and writing sorted buffers to disk continues until the RiverSystem is "empty" and no more records remain in any of the streams. At this point, either the records have been written out in a single sorted run, in which case the sort is complete, or if there was not enough memory for a single pass, the records have been written out in multiple sorted runs.

Merging

The current implementation (as of 8/98) of two-pass sort does not merge records once they have been written to disk as sorted runs. The following is a brief description of how this merging phase would be implemented:

If the records have been written to disk as multiple sorted runs, then they must be merged together and written to disk as a single completed run to complete the sort. A Tournament Sort consists of a tree of records and a number of "round" in the tournament. Every round of the tournament a new record is chosen in the tree as the "winner" which is the least of all the keys of the records participating within the round. The winner is removed from the tree and placed in a buffer, and a new record is read from disk to replace the winners place in the tournament tree. This process continues until the tree is empty.

Performance

In the first stage of the single pass NTClusterSort, the underlying RiverSystem is the only component of the system that is active, and its performance is consistent with the range partition RiverSystem. The second stage of the single pass NTClusterSort is the actual sorting and writing to disk, which must be done sequentially. The total throughput of the two node single pass NTClusterSort is approximately 6 MB/s, in which the CPU is near saturation (85% -

7

100%) during the entire operation. This works out to be 53 cycles per sorted byte. 6 MB/s is approximately twice as fast as NTSort performed on the Pennysort Machine. [Gray]

Between the implementation of the one-pass and two-pass sorts, a decision was made to replace IOCPs, ReadFile and WriteFile with Events, WaitForMultipleObjects, WSASend and WSARecv. We made this alteration to accommodate an early version of an experimental protocol AF_ViA which only supports WSA calls and Events. In making these changes, as well as a code revision, we realized a dramatic performance loss. Although we obviously suspect the loss is due to performance differences between the two sets of functions, it is unclear at this point and warrants further investigation.

5. Remote Execution and Installation

Although installation and execution are key components of any application, they are often trivial and taken for granted. However, in a distributed environment these issues become non-trivial, but nonetheless they are often left as an "exercise to the reader." We feel as thought these two issues are crucial features of any distributed application.

Yes, there are installation packages, but as far as we know there are no *distributed* installation packages currently available. Though it may seem like a small inconvenience to have to manually install the application once on each node of a small cluster, it becomes a sizeable chore to install the application on a medium or large cluster (64 or 128 nodes.) It is also natural to assume that a user would want to reinstall or uninstall the application.

Although there are remote execution environments available, this is a poor excuse not to address the issue for two reasons. First, a user must depend on the remote execution environment to be present in a system, and must be familiar with how to integrate the particular distributed application into it. And second, these environments typically cost money.

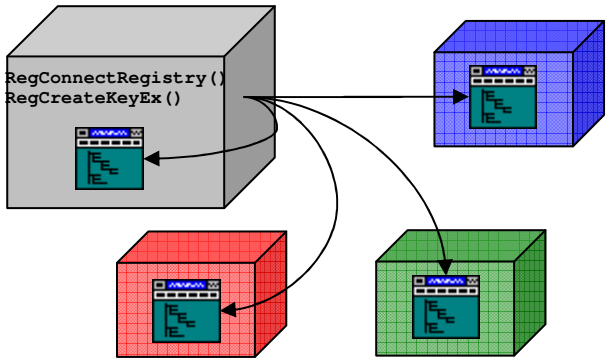

Figure 11. Remote installation consists of editing remote registries via the Win32 RegistryAPI. Adding Key entries that correspond to NTClusterSort COM object allows remote method invocation.

Our solution uses DCOM to remotely instantiate NTClusterSort objects and invoke various methods on these objects to control the distributed system. The installation system is part of the application. The method consists of using the Win32 Registry API to remotely add Registry keys to the nodes in the cluster and using the redirector to copy the binary to the remote disk. Editing the Registry is necessary because in order to instantiate a COM object remotely, certain pieces of information are required to reside in the Registry. Once the Registry entries have been made and the binary has been transferred the installation is complete.

Remote execution of DCOM objects is performed by CoCreateInstanceEx(), which instantiates the object on a remote node and returns a handle to the remote object. Once the DCOM objects are instantiated, object methods, such as InitiateSort(), GetProgress() etc. can be invoked as if they were local.

Although we have built a prototype control GUI to demonstrate remote installation and execution, the prototype is primitive and only demonstrates the feasibility of such a system. For installation we currently implement the remote installation via system call to REGINI, rather than use the Registry API. There are many subtleties of DCOM which must be better understood before we can create a robust remote execution package. Remote execution is limited to one instantiation per node at a time, which limits our ability to run multiple instances on SMPs. We also found that proper shutdown of remote objects is an issue which must be better understood.

6. Summary and Conclusions

We have presented the design and implementation of three distributed systems, DataPump, RiverSystem and NTClusterSort. The *DataPump* uses 75% of the processor and delivers twice the throughput as the default methods of transferring data between nodes. We discussed the Winsock nagling problem on the receive side which consumes 2.5 times as much CPU as the receive side. We attribute this performance win to using overlapped I/O and I/O Completion Ports as our synchronization object. At 9 MB/s, the bottleneck in the DataPump in our experiments was the disks.

We explained the *RiverSystem* system to manage data flows. The system consists of a series of data sources and sinks which produce or consume from the data river through a GetRecord and PutRecord interface. The RiverSystem is intended to be a template for various types of data flows through the PickStream() function which determines which tributary of the river the records are sent to. In our experiements the performance of the RiverSystem is limited by the CPU, with a transfer rate of 7.3 MB/s on a two node system.

We also presented *NTClusterSort* as a sample application based on the RiverSystem that performs distributed one pass and partial two pass sorts. We discussed how a sophisticated sorting algorithm is not necessary to achieve high performance because of the determining factor is overwhelmingly the I/O cost involved. NTClusterSort uses four threads: the Inbound thread, Outbound thread, ExtraAndSort thread and a WriteDisk thread. The Inbound Thread coupled with the ExtractAndSort and WriteDisk threads make up a three stage pipeline to achieve a high level of resource utilization. We reported that the performance of a one pass sort on two nodes sorted data at a rate of 6 MB/s. Although the two pass sort implementation is only partial, we explained how a simple tournament sort could be used to implement the merge phase. We also discussed the possibility of performance degradation in the latest version of NTClusterSort due to a change from ReadFile/WriteFile and IOCPs to WSASend/WSARecv and Events/WaitForMultipleObjects. This warrants further experiments to determine the difference in performance.

We concluded with a brief discussion of a method of remote installation and execution using DCOM. Installation involves using the Registry API to install COM information on remote nodes, and CoCreateInstanceEx() is used to instantiate remote objects.

Our intent was to develop a distributed data management system with source code made available to the public in hopes that it will be configured, adapted and improved upon. In retrospect, the functional programming style we used did not lend itself to the extensibility we envisioned, but it allowed us to focus on developing a high performing prototype system. Initially issues of functionality and performance were paramount, but once the viability of a high performance system was realized, extensibility issues seemed to become most important. Visit http://www.research.microsoft.com/barc for the latest version of the source code.

7. Acknowledgements

We would like to thank Reza Baghai, Wael Bahaa-El-Din and Venkat Ramanathan for their patient and generous assistance to us in understanding TCP performance issues and for the use of their performance lab. Luis Rivera of UCI provided us with a reference point with his NT port of the one pass NOWSort code from UC Berkeley. Catherin van Ingen consulted us on various high performance I/O issues and Jim Gemmel offered useful discussion about the Winsock nagling issue. Don Slutz and Tom Barclay gave useful feedback and comments about the design of the three systems.

9. References

[Riedel] Erik Riedel, Catherine van Ingen, Jim Gray. "Sequential I/O on Windows NT 4.0 – Achieving Top Performance", http://www.research.microsoft.com/barc July, 1998

[Barclay] Tom Barclay, Robert Barnes, Jim Gray, Prakash Sundaresan. "Loading Databases Using Dataflow Parallelsim", SIGMOND RECORD, Vol 23, No. 4 December 1994

[Arpaci-Dusseau] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, David A. Patterson. “High-Performance Sorting on Networks of Workstations” *SIGMOD '97*, Tucson, Arizona, May, 1997

[Gray] Jim Gray, Joshua Coates, Chris Nyberg. “Performance / Price Sort”, http://www.research.microsoft.com/barc July 1998.

10