Expansion of a Framework for a Data-Intensive Wide-Area Application to the Java Language

Sergey Koren

Table of Contents

1 Introduction

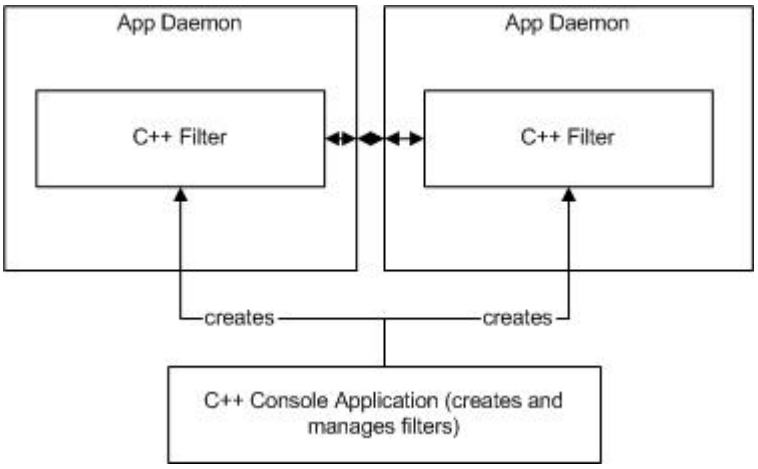
Applications that process large, distributed, data have become increasingly prominent. This dependence on data and the pervasiveness of wide-area networks has led to the development of a filter-programming based application, known as the DataCutter [1]. At the same time, the large set of included libraries and the adherence to object oriented design of the Java language has led to its wide adoption. With the new increases in performance, Java is fast becoming a realistic alternative to C++, even in high-performance application areas. The acceptance of the Java language has created a need for a similar infrastructure that allows its use on a distributed system. This paper presents a development of an expanded filter-stream programming infrastructure to allow distributed applications to be developed in Java. The expanded system focuses on providing both a service to Java development and a merger between the languages of C++ and Java. The paper also presents experimental results demonstrating the influence of Java on the performance of the application.

2 Previous Research

The existing distributed application is known as the DataCutter and was originally developed by Michael D. Beynon at the Department of Computer Science at the University of Maryland, College Park. The DataCutter application is based on the Grid programming approach [1].
It is created to allow several application components, or filters, to run on separate machines and establish communication through streams. The first step is the creation of a daemon to administrate the creation of app daemons. This daemon runs an app daemon on every machine that hosts a filter and serves as the infrastructure between the filters. Communication between filters is restricted to the input and output streams which are provided by a main application [1]. The main application establishes the location of the filters and requests instances of filters from a user-created function by filter name. Each filter is a specialized user class that processes data and returns the result to the requester [1]. A filter consists of three functions, an initialization function called when a filter is created on a machine, a process function which is called repeatedly while data arrives, and a finalize function called when a filter terminates. The console then establishes the means of communication between filters through user-defined parameters and proceeds to launch. The filters perform their tasks and are terminated by the system when

they are done. The filters are built around the stream abstractions [1]. Streams deliver data in fixed-size buffers [1].

The current system can be seen as:



The current application lacks the ability to create Java filters. Since Java has become a popular language, the need for Java filters has grown. The question of the efficiency of Java applications has also been raised. Therefore, the DataCutter application must be expanded to include Java. This will open the path to Java filter creation and the possibility of experimental evidence of Java's performance.

3 Algorithm

3.1 Approach

In order to provide the infrastructure to Java applications, two different approaches were explored. One approach was the development of the infrastructure purely in Java, using the previously developed DataCutter as a reference. An alternative approach was the use of the provided JNI API to allow the Java infrastructure to use existing C++ code. The pure Java infrastructure was ultimately abandoned because of the scope of the required development and the problem of maintaining consistency between both infrastructures. The pure Java infrastructure would also require a separate system to mix of Java and C++ filters in a single application. Therefore, the JNI model was adopted despite several downsides. The first of these was the lack of C++ support in JNI. Ultimately, this was dealt with by developing an emulation

of class structure across languages. A more fundamental problem was the added overhead of the cross-language calls. This was expected to be a reasonable expense, after taking the Java virtual machine creation overhead into account. The prototype uses the JNI capability because the downsides of the alternative outweighed its advantages. It also provides a natural mechanism for combining Java and C++ filters in a single distributed application. In an attempt to address the concerns about performance, the experimental results provide an estimate of the impact of the JNI model.

3.2 C++ Infrastructure Modifications

The development of the Java infrastructure focused on allowing the creation of Java filters that could coexist together with other Java filters and C++ filters. In order to make the Java filters compatible with the existing C++ infrastructure for DataCutter, several additions were required to the existing C++ code.

The first addition was required to allow the C++ infrastructure to create Java filters. To isolate the infrastructure from these changes, a special system filter was created which acts as a C++ version of a Java filter. An instance of this filter exists for each created Java filter on a system and provides a path for calls to travel from the C++ infrastructure to a Java filter. These calls include the requests for filter creation and the invocation of user-implemented filter functions.

The second addition was necessary to allow Java to recognize C++ functions. Functions that will be called by Java must be registered, or recorded, in the Java virtual machine upon creation. Two approaches are possible. The first is the creation of a shared object, containing the C++ infrastructure, which is loaded by the Java filter. The second is the manual loading of the virtual machine by the C++ infrastructure. The approach used was that of manual loading of the virtual machine. This approach is more natural because the Java filters are created by the C++ infrastructure and, when the first Java filter is loaded on a machine, the virtual machine can also be loaded. To isolate the registration of these methods, a singleton class was created to maintain an instance of the created virtual machine and provide access to Java from the C++ code.

3.3 Java Infrastructure

Finally, several classes present in C++ were mimicked in Java to provide necessary DataCutter functionality. As a result, the C++ infrastructure had to be expanded to allow the emulation of class-structure between the languages. JNI allows a Java function to call a corresponding C function, but does not allow state to be maintained in the system. There is no built in way for a JNI function in C++, called from a Java function, to know what instance of a C++ class it is attached to. This functionality is not present in JNI because it was originally developed to interface to C and not C++. The approach used to maintain the state is known as the Registry implementation [2]. The C++ infrastructure creates a singleton hash map that stores a Java class hash code, used as the key, and a pointer to a corresponding C++ class. Each function in the C++ class is copied in the Java emulated class and uses native a call into C++. There, the instance of the C++ class is retrieved based on the hash code passed down from Java and the appropriate function is called on the C++ instance. After the C++ function finishes, any result is returned back to the Java class. In addition, all public member variables are wrapped by functions that forward the calls to cause modification to the C++ data members. The downside of this approach is that a user-implemented hash code function can break the uniqueness of the hash key. However, Java provides a default implementation that is based on memory location of a class instance and allows unique keys, without user intervention. The completed classes provide a Java filter with the necessary functionality to allow it to interact with the existing C++ infrastructure.

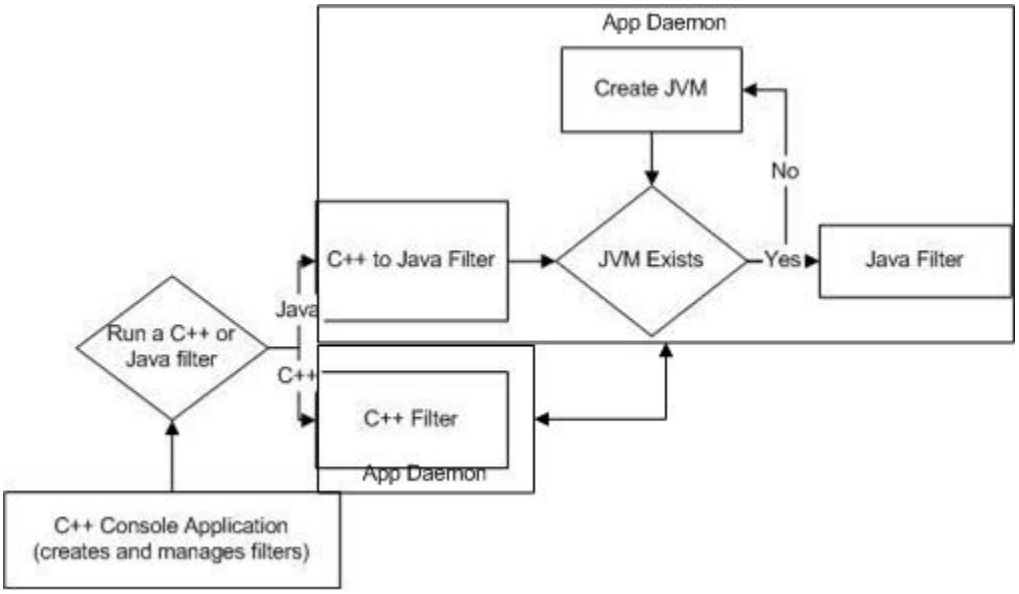
A caveat of this approach is the problem of public class pointers or references in C++ contained within other classes. In this case, it is not enough to simply provide the functions to wrap the class member and forward calls. The problem stems from the fact that in order to return a result class in Java a new instance would have to be created every time a get function is called. This happens because the Java class would have no information about an already existing instance. The problem impedes user functionality by invalidating all previously returned Java class instances. That is, a user may call get once and save the reference, and, upon the next call to get, the user's previous reference becomes invalidated because its connection to the C++ infrastructure is overwritten by the new instance. To avoid this and the performance impact of unnecessary instance creation, whenever a C++ class stores a pointer or reference to another

class, the Java mimicking class also holds a corresponding reference. Whenever appropriate the Java class's reference is updated to ensure its C++ connection is consistent.

Together, these modifications allow the Java filter to interact with other filters, C++ or Java through the existing infrastructure. No changes are required to the existing filters to allow the addition of Java filters. Any changes would be made to the main function, where the Java filters and their connections will be listed. Therefore, the Java filters are isolated from the existing code to allow maximal flexibility in modification of each part separately of the other. The Java filters will also automatically benefit from any enhancements made in the C++ infrastructure and any added functionality can be easily emulated in the corresponding Java classes.

3.4 System Overview

The resulting system can be viewed as the following:



The main console application, which controls the filters, remains in C++, as does the app daemon. The user must provide a function that returns an instance of a filter based on a string name. The added C++ infrastructure allows a user to call a single function to see if a given string is a valid Java filter, returning an error if a failure occurs. Whenever a Java filter is created, the C++ infrastructure verifies the existence of the Java virtual machine, and creates an instance of the appropriate Java filter. The creation of the Java filter illustrates the up-calls to the Java virtual machine. The Java filter has three functions, start_work, called on filter creation,

process, called repeatedly when data arrives, and finish_work, called when a filter is destroyed. The emulated classes in Java provide the filter access to the stream abstraction through down-calls to the infrastructure.

4 Results

4.1 Performance Testing

The performance experiments were done on the hyena.cs.umd.edu and condor.cs.umd.edu.

Machine Name	Machine Type	OS	Machine Type
hyena.cs.umd.edu	Sparc	5.6 Generic_105181-21	Ultra-5_10
condor.cs.umd.edu	Sparc	5.6 Generic_105181-21	Ultra-1

All experimental results were created using milliseconds as the standard unit of time measure. The timeval structure was used to calculate elapsed time in C++ and System.getCurrentTimeMillis() was used to calculate elapsed time in Java.

The first experiment was designed to find the overhead of the creation of all the necessary objects for a Java filter. Therefore, an empty filter was written in both C++ and Java and was created on hyena.cs.umd.edu.

Filter Language	First Run	Subsequent Runs
Java	1560	400
C++	0.02	0.02

As expected, the creation of a Java filter has a lot of overhead. However, a Java filter's creation time is cut dramatically after the first run. This is most likely due to the optimizations by the Java virtual machine.

In order to determine what was causing the large delays in Java filter creation, an experiment was developed where two filters were created on a single machine. This experiment was designed to determine how much time is used by the creation of static structures, such as a virtual machine. The filters were once again created on hyena.cs.umd.edu.

Filter Language	First Run		Subsequent Runs	
	1st Filter	2nd Filter	1st Filter	2nd Filter
Java	1540	360	400	140
C++	0.013	0.013	0.013	0.010

The creation of the JVM (Java Virtual Machine) on the first run took 1200 milliseconds on the first filter. Upon further experimentation, it was discovered that 250 milliseconds was used to actually create the appropriate Java class from C++. This result implies that there is a large delay for JNI calls from C++ into Java that impedes Java filter creation speed. However, both the creation of the JVM and the calls into Java are optimized on second runs, with the JVM creation dropping to only 240 milliseconds. To determine the cost of forwarding calls to the appropriate filter functions in a Java a test used a filter with all the functions blank was used.

Function Called	Time Taken
init	.5
process	.5
finalize	.2

As can be seen here, there is some performance impact of calling the Java functions, but it is reasonable.

Having finished an evaluation of individual runtimes, the impact of Java on inter-filter communication was examined. Since the main function of filters is stream input and output, the experiment included an evaluation of read and write calls in both C++ and Java filters. The experiment consisted of running two filters on two machines, with one filter being a producer and one a consumer. Each run of the filter sent several buffers of varied size in bytes. In order to better evaluate performance, all pairs of C++ and Java connections were tested and the filters were switched between the two machines.

The tables below indicate the results:

Producer Language	Producer Location	Consumer Language	Consumer Location	Buffer Size	Write Results	Read Results
Java	hyena.cs.umd.edu	Java	condor.cs.umd.edu	10	2	3
Java	hyena.cs.umd.edu	C++	condor.cs.umd.edu	10	1	1
C++	hyena.cs.umd.edu	Java	condor.cs.umd.edu	10	0.1	3
C++	hyena.cs.umd.edu	C++	condor.cs.umd.edu	10	0.1	1
Java	condor.cs.umd.edu	Java	hyena.cs.umd.edu	10	2	2
Java	condor.cs.umd.edu	C++	hyena.cs.umd.edu	10	3	0.5
C++	condor.cs.umd.edu	Java	hyena.cs.umd.edu	10	0.2	2
C++	condor.cs.umd.edu	C++	hyena.cs.umd.edu	10	0.2	0.07

Producer Language	Producer Location	Consumer Language	Consumer Location	Buffer Size	Write Results	Read Results
Java	hyena.cs.umd.edu	Java	condor.cs.umd.edu	100	0	1
Java	hyena.cs.umd.edu	C++	condor.cs.umd.edu	100	0	0.1
C++	hyena.cs.umd.edu	Java	condor.cs.umd.edu	100	0.06	1
C++	hyena.cs.umd.edu	C++	condor.cs.umd.edu	100	0.06	0.1
Java	condor.cs.umd.edu	Java	hyena.cs.umd.edu	100	0	1
Java	condor.cs.umd.edu	C++	hyena.cs.umd.edu	100	0	0.06
C++	condor.cs.umd.edu	Java	hyena.cs.umd.edu	100	0.1	0
C++	condor.cs.umd.edu	C++	hyena.cs.umd.edu	100	0.1	0.06

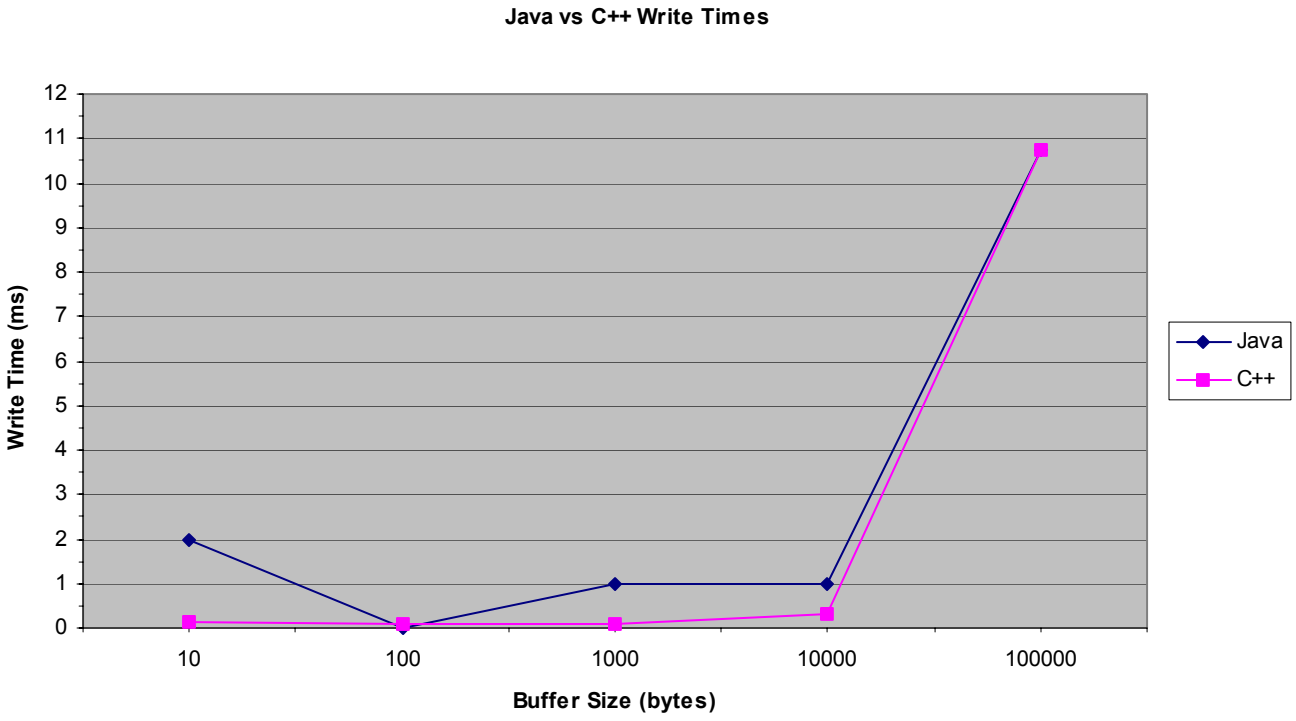
Producer Language	Producer Location	Consumer Language	Consumer Location	Buffer Size	Write Results	Read Results
Java	hyena.cs.umd.edu	Java	condor.cs.umd.edu	1000	1	1
Java	hyena.cs.umd.edu	C++	condor.cs.umd.edu	1000	1	0.1
C++	hyena.cs.umd.edu	Java	condor.cs.umd.edu	1000	0.07	1
C++	hyena.cs.umd.edu	C++	condor.cs.umd.edu	1000	0.07	.01
Java	condor.cs.umd.edu	Java	hyena.cs.umd.edu	1000	1	1
Java	condor.cs.umd.edu	C++	hyena.cs.umd.edu	1000	1	0.06
C++	condor.cs.umd.edu	Java	hyena.cs.umd.edu	1000	0.1	0
C++	condor.cs.umd.edu	C++	hyena.cs.umd.edu	1000	0.1	0.06

Producer Language	Producer Location	Consumer Language	Consumer Location	Buffer Size	Write Results	Read Results
Java	hyena.cs.umd.edu	Java	condor.cs.umd.edu	10000	1	0
Java	hyena.cs.umd.edu	C++	condor.cs.umd.edu	10000	1	0.1
C++	hyena.cs.umd.edu	Java	condor.cs.umd.edu	10000	0.2	1
C++	hyena.cs.umd.edu	C++	condor.cs.umd.edu	10000	0.2	0.2
Java	condor.cs.umd.edu	Java	hyena.cs.umd.edu	10000	1	0
Java	condor.cs.umd.edu	C++	hyena.cs.umd.edu	10000	1	0.06
C++	condor.cs.umd.edu	Java	hyena.cs.umd.edu	10000	0.5	0
C++	condor.cs.umd.edu	C++	hyena.cs.umd.edu	10000	0.4	0.06

Producer Language	Producer Location	Consumer Language	Consumer Location	Buffer Size	Write Results	Read Results
Java	hyena.cs.umd.edu	Java	condor.cs.umd.edu	100000	11	1
Java	hyena.cs.umd.edu	C++	condor.cs.umd.edu	100000	11	0.1
C++	hyena.cs.umd.edu	Java	condor.cs.umd.edu	100000	12	0
C++	hyena.cs.umd.edu	C++	condor.cs.umd.edu	100000	11	0.2
Java	condor.cs.umd.edu	Java	hyena.cs.umd.edu	100000	11	0
Java	condor.cs.umd.edu	C++	hyena.cs.umd.edu	100000	10	0.06
C++	condor.cs.umd.edu	Java	hyena.cs.umd.edu	100000	10	0
C++	condor.cs.umd.edu	C++	hyena.cs.umd.edu	100000	10	0.06

The experiment shows several interesting results. The first is that the cost to read and write the first buffer is greater than subsequent ones, even though it is the smallest in size. This increase in performance is present in both C++ and Java, with the difference being greater Java. As a result, the performance difference between Java and C++ decreases after the first buffer.

A summary of Java versus C++ write times is shown below:



The graph above shows the change in write times in C++ starting from the second buffer to the last versus the Java write times. The performance of each buffer is the average of all writes for that size by a given language. The C++ filter remains faster than Java until the last buffer when a severe slowdown in the underlying filter library evens out the performance. It can therefore be seen that the performance of Java eventually matches that of C++ in writing when the buffer size is large enough, implying that there is a flat cost present in all Java writes that is eventually balanced out by the per-byte cost.

The read times for Java show greater fluctuations than those in C++. The reason for this is that there is no timer with a finer grain than one millisecond in Java. Therefore, when times for Java reads are less than one millisecond, they are rounded down to zero. The frequency of zero and one millisecond reads is approximately equal. Consequently, Java reads take approximately 0.5 milliseconds while C++ reads take approximately 0.1 milliseconds. Since the read times are not affected by the size of the buffer as the write times, the Java performance does not match that of C++ because of the flat cost of the JNI calls.

5 Conclusions

The provided Java infrastructure allows a user to create an application consisting of both the Java and C++ filters, giving maximal flexibility. As expected, the major time-sinks in the running of Java filters are the cross-language calls and the creation of a Java virtual machine. Attempting to decrease the number of JNI interactions between the languages can improve the application performance. There is also the need for the development of more Java filter applications to quantify the performance of Java on different platforms and applications.

6 References

[1] Michael D. Beynon, Tahsin Kurc, Alan Sussman, Joel Saltz. Design of Framework for Data-Intensive Wide-Area Applications. In Proceedings of the 9th Heterogeneous Computing Workshop (HCW2000), May 2000

[2] Rob Gordon. Essential JNI. Prentice Hall, NJ, 1998.