Generalized Distributed Garbage Collection

Chua, Neil
2401 Taft Avenue, Manila 1004,
National Capital Region
catchlines@yahoo.com

Lai, Francis
2401 Taft Avenue, Manila 1004,
National Capital Region
fplfrancispelai@yahoo.com

Mondejar, Jeffrey
2401 Taft Avenue, Manila 1004,
National Capital Region
jmmonde@yahoo.com

Samson, Briane
2401 Taft Avenue, Manila
1004, National Capital Region
bvsam2003@yahoo.com

Tan, Kent
2401 Taft Avenue, Manila 1004,
National Capital Region
cant_10@yahoo.com

ABSTRACT
In today's time, more and more complicated programs are being released; from the traditional, centralized client server type to the distributed environmental one, they create further problems such as managing resources, accessing concurrency, incidence of communication infrastructure, deprivation of security and privacy. This is for the reason that not all resources that are being made may last for an extended time. Eventually, some of them may become idle and would no longer be in use therefore leasing distributed garbage collection to come in. There will be an implementation of distributed garbage collection but will frequently do not perform well in a disseminated manner and will only focus to a specific environment. Thus, a new algorithm was formulated by analyzing the best features of the current use of distributed garbage collection method. The features were then compiled into producing an improved system. After implementing and testing the generalized distributed garbage collection algorithm, results show that based on performance in terms of processing time, it runs in linear time as opposed to the exponential time of available algorithms.

Keywords
Garbage collection, distributed system, multi threading

1. INTRODUCTION
In today's time, there had been a lot of emergence in a lot of things especially in the software and technology department and as these technologies emerge, many solutions has been created and at the same time a lot of problems arises, as one closes another opens. One of the most popular architecture in the software development is the distributed architecture which came from server client architecture and a lot of problems or optimization problems have been created by this architecture. One of these problems is garbage collection. Today there has been many implementation and research about garbage collection but usually the implementation of garbage collection in distributed manner have several drawbacks, one of which is that the implementation is an adapted algorithm that it is usually specified to a given environment and/or language and the other problem is that most of the implementation does not work well in a distributed manner [2]. In light of this, a general or standard algorithm for distributed garbage collection was formulated based on existing algorithms and by doing this we would like to identify best features of different algorithm, also try to identity their disadvantages and compare it to other algorithm.

Several distributed garbage collection algorithms have been formulated for different programming languages applying different methodologies [2]. However, there has not been a distributed garbage collection algorithm which can be used for all programming languages [6]. This research then focuses on formulating a general or standard distributed garbage collection algorithm which can be implemented on most programming languages. In order to do this, only the best features from the different algorithms identified were chosen. The selection was based on a criterion which can be found in Chapter 3. The standard algorithm is a mash up of these best features.

In a distributed environment, global and local distributed garbage collection algorithms must be implemented. In this research, the focus would be on a standard local algorithm. It is designed on a shared-memory multiprocessor. The programming language used to test the algorithm is Java.

2. DANGLING POINTER AND MEMORY LEAKS
Two of the most annoying errors involve in the computer system and operations are the unreclaimed memory (memory leaks) and premature reclamation (dangling pointers) [5] [17].

2.1 Dangling Pointers
A dangling pointer is a reference to storage that is no longer allocated. Dangling pointers are malicious because they seldom crash the program until long after they have been created, which makes them difficult to trace [4]. They come about when programmers create, utilize and then free an object in memory but the object's pointer value does not change, such case is a null pointer. Rather, the pointer is pointing to the de-allocated memory location. Thus the term "dangling" since it points to memory that may no longer hold a valid object [7].

C++, an object-oriented programming language that does not rely on garbage collection makes it easy to create dangling pointers. Here are some examples [4]:

```
delete [] s1;
```

```
delete [] s2;
return f (s1, s2); // s1 and s2 are
dangling pointers
```

This code will probably appear to work unless f or one of the functions that are called during the activation of `f` happen to allocate heap storage. When the bug does show up, it will probably look like a bug in `f` or in one of the functions that `f` calls.

```
typedef Foo_ * Foo;

Foo newFoo (char * x) {
    Foo_ tmp(x);
    return &tmp;
}
```

This is the classic technique for creating a dangling pointer in C.

```
typedef char * Foo;

Foo newFoo (char * x) {
    Foo tmp = new char [strlen (x) +1] ;
    strcpy (tmp, x);
    delete [] x;
    return tmp;
}
```

Here `newFoo` creates a dangling pointer by deleting the client's C-style string.

```
typedef char * Foo;
Foo newFoo (char * s) {
    return s;
}
```

If `newFoo` is supposed to return a `Foo` whose lifetime is independent of the lifetime of its argument, then a dangling pointer will be created when a client deletes the C-style string that was passed to `newFoo`. The bug might appear to lie in the client code, but `newFoo` would be the real culprit.

```
class Foo {
  public:
    Foo (char * x) : len(strlen(x)),
name(x) { }
  private:
    int len;
    char * name;
};

Foo newFoo (char * s) {
    return Foo(s);
}
```

Once again, a dangling pointer will be created when a client deletes the C-style string that was passed to `Foo` or `newFoo`.

```
class Foo {
  public:
    Foo (char * x) {
        len = strlen (x);
        name = new char[len + 1];
        strcpy (name, x);
    }
    virtual ~Foo () {
        delete [] name;
    }
  private:
    int len;
    char * name;
};

Foo newFoo (char * s) {
    Foo foo = Foo(s);
    return foo;
}
```

This code fixes the previous bug by introducing three new bugs. The most obvious is that the compiler inserts an implicit call to `foo.~Foo()` when `newFoo` returns. This implicit call deallocates `foo.name`. Hence the `Foo` that is returned by `newFoo` always contains a dangling pointer.

The other bugs are illustrated by the following client code:

```
Foo f1 = newFoo ("hi there");
Foo f2 = f1;
Foo f3;
f3 = f2;
```

Since no copy operator is defined, the compiler will implicitly define a copy constructor that makes `Foo f2 = f1` roughly equivalent to:

```
Foo f2;
f2.len = f1.len;
f2.name = f1.name;
```

Thus `f2.name` becomes the same pointer as `f1.name`. Similarly, no assignment operator is defined, so the compiler will implicitly define an assignment operator that makes `f3 = f2` roughly equivalent to

```
f3.len = f2.len;
f3.name = f2.name;
```

Thus each of `f1`, `f2`, and `f3` contain exactly the same pointer. When they go out of scope, that pointer will be deallocated not once, but three times.

A storage leak would be created if we were to remove the destructor or to remove the call to delete, so those are not good alternatives. What we need is a copy constructor and an overloaded assignment operator.

```
class Foo {
  public:
    Foo (char * x) {
        len = strlen (x);
        name = new char[len + 1];
        strcpy (name, x);
    }

    virtual ~Foo () {
        delete [] name;
```

```
    }

    Foo     (const     Foo     &     foo);
// copy constructor
    const Foo & Foo:operator= (const Foo
&);   // assignment operator
  private:
    int len;
    char * name;
};

//  copy constructor
Foo::Foo (const Foo & foo) {
    len = foo.len;
    name = new char [foo.len];
    strcpy(name, foo.name);
}

//  assignment operator
const Foo & Foo::operator= (const Foo &
rhs) {
    delete [] name;
    name = new char [rhs.len + 1];
    strcpy(name, rhs.name);
    return                          *this;
// so x = y = z will work
}

Foo newFoo (char * s) {
    Foo foo = Foo(s);
    return foo;
}
```

This code still contains a bug. Consider the client code:

```
  Foo f1 = newFoo ("hello");
    Foo f2 = newFoo ("goodbye");
    f1 = flag ? f1 : f2;
```

The assignment represents an implicit call to `f1.operator=(flag ? f1 : f2)`. Suppose flag is true, so the value of the right hand side of the assignment is a reference to `f1`. The code for `f1.operator=` begins by deleting `f1.name`. It then passes the dangling pointer `f1.name` as both arguments to `strcpy`. Following the assignment, `f1` contains a dangling pointer. When `f1` goes out of scope, and its destructor is called, the `delete []` operator will be called on `f1.name` for the second time.

The solution for this problem is to make the assignment operator check whether this is equal to the right hand side:

```
const Foo & Foo::operator= (const Foo &
rhs) {
    if (this == &rhs) {
        delete [] name;
        name = new char [rhs.len + 1];
        strcpy(name, rhs.name);
    }
    return                          *this;
// so x = y = z will work
}
```

2.2 Memory Leaks

Memory leak is another problem that may occur in the computer system that leads to poor performance and failure. Memory leak is when the system does not manage its memory allocation properly [3]. When you forget to free a block of memory allocations with the operator, say, new then memory leaks occur. This will lead to application's run out of memory and may even cause the system to crash. Here are some examples [12]:

First, delete it before reallocating it.

```
char *string;
string = new char[20];
string = new char[30];
delete [] string;
```

In this example, there is the new and delete operations. The goal is to find the leakage. Noticeably there are two consecutive memory allocations by means of the string pointer, but something seems to be missing. There should be a `delete []` statement right after the first allocation and then try to reallocate using a different size parameter. If this is not done, the second allocation will assign a new address to the string pointer while the previous one will be lost. This makes it impossible to free the first dynamic variable further on in the code, resulting in a memory leakage.

Second, a pointer to each dynamic variable must exist.

```
char *first_string = new char[20];
char *second_string = new char[20];
strcpy(first_string, "leak");
second_string = first_string;
delete [] second_string;
```

This example shows a memory leak. In detail, the address of the dynamic variable associated with `second_string`, as a side-effect of the pointer assignment, was lost so it cannot be deleted from the heap anymore. Thus the last line of code only frees the dynamic variable associated with `first_string`, which is not desired.

The main idea is to try and not lose the addresses of dynamic variables as one may eventually not be able to free them.

Lastly, monitor local pointers.

```
void leak() {
    int k;
    char *cp = new char('E');
}
```

Noticeably, both the k and cp variables are local so they are allocated on the stack segment. Then when it comes the time to exit the function, they will be freed from memory as the stack is restored. But the dynamic variables associated with the cp pointer were not erased from heap at function exit.

2.3 Mark and Sweep Garbage Collection

Mark and sweep is the first garbage collection technique for automatic storage reclamation [11]. Using this, unreferenced objects are not reclaimed immediately. Rather, they were accumulated as garbage, undetectable and unreachable until all

available memory has been exhausted [7] [15]. In doing so, the execution of the program will be temporarily suspended until all unreferenced objects are reclaimed. Then the execution of the program will be resumed [15].

Mark and sweep is known as the tracing garbage collector because it will exhaustedly trace out the collection of object whether directly or indirectly accessible by the program. All accessible objects are referred to be live. All inaccessible objects are known as garbage.

The algorithm has two phases: mark phase and the sweep phase. In the first phase, it marks all the accessible objects. In the second phase, it scans through the heap and reclaim all unmarked objects (garbage) [15].

3. SPECIFIC CRITERIA USED

These are the specific criteria used in the evaluation of the different features in a distributed garbage collection algorithm.

First would be safety. In this criterion, only garbage should be reclaimed. Next would be that the collection should be complete. All objects that are garbage at the start of the garbage collection cycle should be reclaimed by its end. In particular, it should be possible to reclaim distributed cycles of garbage. Third would be concurrency. Distributed GC should not require the suspension of mutator or local collector processes; distinct distributed garbage collection processes should be able to run concurrently.

Efficiency should also be considered in evaluating features. Garbage should be reclaimed promptly and without delay. Another criterion would be expediency. Whenever possible, garbage should be reclaimed despite the unavailability of parts of the system. Next would be scalability. Distributed GC algorithms should scale to networks of many processes. Lastly, the feature must be fault tolerant. Memory management system should be robust against message delay, loss or replication or process failure.

4. GENERALIZED DISTRIBUTED GARBAGE COLLECTION

As mentioned, this new algorithm made use of different features, namely: time-to-live, clustering heaps, mark and sweep, and cycle detection. These features are then put strategically into different modules which will be discussed in this section.

4.1 Object Creation Module

This module has three phases namely, object creation, run time and, object registration.

On object creation of a certain process, the object will have a parameter to determine the lifespan of the object. The lifespan may vary according to the type of objects created.

The lifespan of an object will be determined by its Time-to-Live(TTL) parameter. The TTL value will determine if the object is ready for garbage collection or not.

The object will also have an extra parameter called “Marked/Unmarked” to be used by the Mark and Sweep Module.

After object creation, objects that are created will have to be registered in order to determine their location on garbage collection.

Lastly, after registration, the objects will run their course for what they were meant to do. The objects will terminate on their own when their TTL time runs out.

4.2 Garbage Detection Module

In this phase, the system undergoes three processes namely, garbage detection, TTL timer detection and clustering.

This module starts off by simultaneously detecting garbage objects with all the programs in a simulated distributed setting. In detecting garbages, it starts by determining which among the objects has an expired TTL timer. In the event that the TTL timer has expired, the garbage detection and collection process will start. After detecting which are garbage, the algorithm must now determine which among the objects are ready for garbage collection. To do this, the objects are clustered into heaps determined by their common attributes. Objects that share the same set of resources are clustered into one heap, objects that have expired TTL timers are clustered in another and so on.

4.3 Garbage Collection Module

In the last phase, the system undergoes marking, cycle detection, sweep and TTL timer refresh.

In this module, the system scans the set of heaps to determine which objects are ready for collection, those objects that are unreachable, by marking the objects parameter “Marked/Unmarked” into marked. It then determines if there are objects that reference one another in a cyclic manner. These objects are not referenced by any other objects but by themselves thus they are considered to be garbage. Upon detection, the Cycle Detection marks the parameters of these object for collection. After marking and cycle detection, the system will now then pass through the heaps to collect all marked objects by sweeping. And upon collection, all of the remaining objects are considered to be alive and non-garbage objects. Thus their TTL timers are refreshed and the entire garbage detection and collection starts again.

5. ANALYSIS

There are different ways to test and verify the algorithm for distributed garbage collection, in this research we will be using server client approach, token ring approach and the barrier sync approach [9].

In the client server approach, a server generates a resource and randomly assigns it to any client. The server generates random amount of resources per interval and assigns them with random amount of time to live parameter. The server also randomly renews the time to live value of random resources and link them randomly to be able to create a cycle. The algorithm will have to detect if the allocated resources have become garbage and detects if there are cyclic garbages within them [9].

Another approach in testing the distributed garbage collection is the token ring approach. The computers are connected in a ring topology and whenever an idle computer gets a hold on the token, it will generate its own set of resources that will have random amounts of time to live values. Random instances of

resources will be connected across the network and will have cyclic attributes. These resources will be able to communicate with each other and determine which among the resources are ready for garbage collection [9].

Lastly the barrier sync can also be used for testing, wherein an enormous number of processes are created and have a short time to live parameter and the idea is that the algorithm must be able to process many instances of the same process and be able to reclaim them as soon as possible, this behavior is random when distributing the processes and is a good test for the algorithm. This approach enables the system to be tested for its handling capacity. Since the system will be overloaded with a huge number of resources to be collected at the same time, we will be able to benchmark its capabilities and its failing points [9].

The theoretical framework is implemented in Java using threads. It is composed of two primary running threads which is the (i) Object Creator and the (ii) Garbage Collector. Threads were used in order to simulate individual running processes on different processor cores.

The Object Creator indefinitely generates random threads (Objects) simulating different applications. Each object has their own Time-to-Live parameter as well as attributes to determine related characteristics. The Garbage Collector performs the necessary procedures as discussed in the theoretical framework to perform garbage collection. The memory is represented as a fixed size ArrayList of Objects.

The system is set to run on different scenarios wherein a fixed maximum number of objects are created and the system is tested for its completeness meaning if it can successfully detect, classify and collect garbage objects. For the scenario, the system ran on 1000 to 5000 object creations with a memory of 500 objects simultaneously running and an average time to live of objects at 30 seconds. Note that these numbers are randomly chosen especially the time to live parameters.

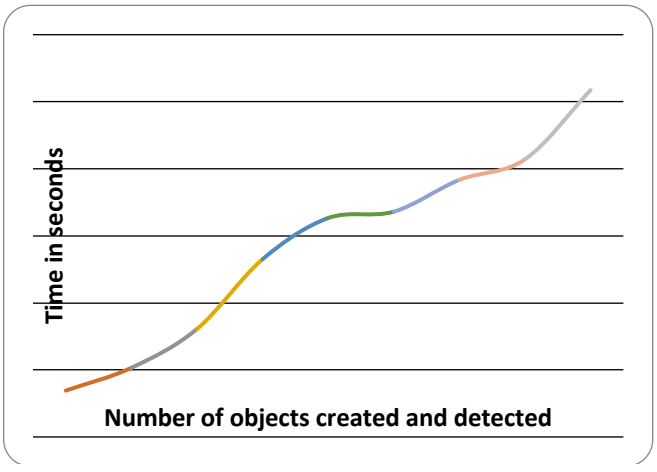

Figure 1 Overall system performance

Based on our test as shown in figure, our system is able to run in a linear fashion such that as the number of objects created and deleted and detected increase, the processing time also increase by the same amount. Figure 1 shows irregularities in the linear movement of the graph. These are due to the performance drops with certain values of objects created, but overall our system is capable of performing just like what other algorithms are capable of and maybe more.

6. CONCLUSION AND FUTURE DIRECTIONS

In this study, we have proposed a new algorithm in distributed garbage collection. This algorithm has been simulated into an environment and compares the results with the other algorithm. Based from the result the new algorithm performed better and is a general algorithm which can in turn be implemented to different settings. Overall we have achieved our goal of creating a general algorithm for distributed garbage collection.

For future work, it is recommended to test the system under different scenarios and system setups so that the irregularities can be further explained. Since the algorithm aims to provide a generic or standard means of garbage collection in a distributed environment, it would be very beneficial if the algorithm will be tested in a wide variety of distributed environments and not just on what has been used. Also, including other garbage collection methods or algorithm features may further improve the performance of our system. Though the algorithm has already showed an improved performance as opposed to existing algorithms, there may still be useful algorithm features and garbage collection methods which are left undiscovered.

References

[1] Cobb, Michael (2007). How to avoid dangling pointers: Tiny programming errors leave serious security vulnerabilities. Retrieved November 17, 2009 from http://searchsecurity.techtarget.com/tip/0,289483,sid14_gci1271770,00.html

[2] Cohen, M., Kooi, S. & Srisa-an, W. (2006). Clustering the Heap in Multi-Threaded Applications for Improved Garbage Collection. Proceedings of the 8th annual conference on Genetic and evolutionary computation, pp. 1901 – 1908.

[3] Crockford, Douglas. (n.d.). JScript Memory Leaks. Retrieved November 17, 2009 from http://javascript.crockford.com/memory/leak.html

[4] Dangling pointers. Retrieved November 17, 2009 from http://www.ccs.neu.edu/home/will/com1205/dangling.html

[5] Görtz, Thorsten. The "More for C++" Garbage Collector. Retrieved November 17, 2009 from http://www.morefor.org/documentation/gc.html

[6] Grossman, D. (2007). The Transactional Memory / Garbage Collection Analogy. Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, pp. 695 – 706.

[7] Jones, Richard and Lins, Rafael (1999). Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley and Sons Ltd.

[8] Kapadia, V. & Thakore, D. (2009). Distributed Garbage Collection Using Client Server Approach in Train Algorithm. Proceedings from the 2009 IEEE International Advance Computing Conference, pp 492 - 495.

[9] Klintskog, E. (2005). Component-Based Distributed Garbage Collection. ACM Transactions on Programming Languages and Systems (TOPLAS),

[10] Knobe, K., Harel, N. & Mandviwala, H. (2006). Distributed Garbage Collection Algorithms for Timestamped Data. IEEE Transactions on Parallel and Distributed Systems, 17 (10), 1057-1071.

[11] McCarthy, John (1960). Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, 184-195. Retrieved November 17, 2009 from http://www-formal.stanford.edu/jmc/recursive.html

[12] Memory leaks in C++ and how to avoid them. Retrieved November 17, 2009 from http://www. codersource.net/c++_memory_leaks.aspx

[13] Ning, Z., Zhang, C., Yang Xia, G. X., "A Hybrid Distributed Garbage Collection of Active Objects." IEEE International Conference on Embedded Software and Systems Symposia, 2008, pp. 13 – 16.

[14] Pizlo, F., Frampton, D. & et al. (2007). STOPLESS: A Real-Time Garbage Collector for Multiprocessors. Proceedings of the 6th international symposium on Memory management, pp 159-172.

[15] Priess, Bruno R. (n.d.). Mark-and-Sweep Garbage Collection. Retrieved November 17, 2009 from http://www. brpreiss.com/books/opus5/html/page424.html

[16] Sung-Wook Ryu, Eul Gyu Im & Clifford Neuman, B.(2003). Distributed Garbage Collection by Timeouts and Backward Inquiry. IEEE Computer Society Proceedings of the 27th Annual International Conference on Computer Software and Applications Page: 426

[17] Veiga, L. & Ferreira, P. (2005). Asynchronous Complete Distributed Garbage Collection. Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, pp 1-10.

[18] Veiga, L., Pereira, P. and Ferreira P. (2007). Complete distributed garbage collection using DGC-consistent cuts and .NET AOP-support. IET Softw., Vol. 1 No. 6, 263 - 279. Retrieved September 24, 2009 from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4435105 &isnumber=4435100

[19] Wang, W. and Varela C., "Distributed Garbage Collection for Mobile Actor Systems: The Pseudo Root Approach," In GPC'06. Springer-Verlag, 2006.