Real Time Cloud Computing

Nitesh Kumar Jangid
Amity Institute of Information Technology, Amity University Rajasthan, Jaipur, Rajasthan, India
njangid@jpr.amity.edu

ABSTRACT
With the development of parallel computing, distributed computing, grid computing, a new computing model appeared. The concept of computing comes from grid, public computing and SaaS. The vision of cloud computing is to reduce the cost of computing, increase reliability, and increase flexibility by transforming computers from something that we buy and operate ourselves to something that is operated by a third party.

Real-time application distinguishes itself from general-purpose application by imposing time restrictions on parts of the runtime behavior. Such restrictions are typically placed on sections of the application such as an interrupt handler, where the code responding to the interrupt must complete its work in a given time period.

It is possible to realize a Real Time Cloud Computing where applications in cloud will respond within a specified time frame. If some Real Time Clouds will be developed then it may be possible to manage a disaster sites more efficiently with more intelligent robots that ever before, help hospitals to cure their critical patients within specified time frame, etc.

This paper also introduces a model to develop the applications for Real Time Cloud using cutting age technologies like Java(RTJS) and Metronome GC.

KEYWORDS
Cloud Computing, Real Time(RT), Garbage Collector(GC), Real Time Java Specification(RTJS), Graphical User Interface(GUI), Java Virtual Machine(JVM), Real Time Garbage Collector(RTGC), Infrastructure as a Service(IaaS), Platform as a Service(PaaS), Software as a Service(SaaS), EC2(Elastic Cloud Computing), S3(Simple Storage Service), Customer Relationship Management(CRM), Process, RAM(Random Access Memory/Read Write Memory), Memory Leak, Stop-the-world(STW), Application(APP), Incremental GC, Sliding Window, NoHeapRealtimeThreads(NHRTs) & Real Time Cloud(RTC).

INTRODUCTION
The Cloud is a large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet.

The vision of Cloud Computing is to reduce the cost of computing, increase reliability, and increase flexibility by transforming computers from something that we buy and operate ourselves to something that is operated by a third party. Governments, research institutes, and industry leaders are rushing to adopt Cloud Computing to solve their ever increasing computing and storage problems arising in the Internet Age. There are three main factors contributing to the surge and interests in Cloud Computing:

1. Rapid decrease in hardware cost and increase in computing power and storage capacity, and the advent of multi-core architecture and modern supercomputers consisting of hundreds of thousands of cores.
2. The exponentially growing data size in scientific instrumentation/simulation and Internet publishing and archiving.
3. The wide-spread adoption of Services Computing and Web 2.0 applications.

RT application development distinguishes itself from general-purpose application development by imposing time restrictions on parts of the runtime behavior. Such restrictions are typically placed on sections of the application such as an interrupt handler, where the code responding to the interrupt must complete its work in a given time period. When hard RT systems, such as heart monitors or defense systems, miss these deadlines, it's considered a catastrophic failure of the entire system. In soft RT systems, missed deadlines can have adverse effects -- such as a GUI not displaying all results of a stream it's monitoring -- but don't constitute a system failure.

It is possible to realize a Real Time Cloud Computing where applications in cloud will respond within a specified time frame. If some Real Time Clouds will be developed then it may be possible to manage disaster sites more efficiently with more intelligent robots that ever before, help hospitals to cure their critical patients within specified time frame, etc.

In Java applications, the JVM is responsible for optimizing the runtime behavior, managing the object heap, and interfacing
with the operating system and hardware. Although this management layer between the language and the platform eases software development, it introduces a certain amount of overhead into programs. One such area is GC, which typically causes nondeterministic pauses in the application. Both the frequency and length of the pauses are unpredictable, making the Java language traditionally unsuitable for RT application development. Some existing solutions based on the Real-time Specification for Java (RTSJ) let developers side step Java technology’s nondeterministic aspects but require them to change their existing programming model.

Metronome is a deterministic garbage collector that offers bounded low pause times and specified application utilization for standard Java applications. The reduced bounded pause times result from an incremental approach to collection and careful engineering decisions that include fundamental changes to the VM. Utilization is the percentage of time in a particular time window that the application is permitted to run, with the remainder being devoted to GC. Metronome lets users specify the level of utilization an application receives. Combined with the RTSJ, Metronome enables developers to build software that is both deterministic with low pause times and pause-free when timing windows are critically small.

This paper explains the limitations of traditional GC for RT applications, details Metronome’s approach, and presents tools and guidance for developing hard RT applications with Metronome.

CLOUDS, GRIDS, AND DISTRIBUTED SYSTEMS
We argue that Cloud Computing not only overlaps with Grid Computing, it is indeed evolved out of Grid Computing and relies on Grid Computing as its backbone and infrastructure support. The evolution has been a result of a shift in focus from an infrastructure that delivers storage and compute resources (such as the case in Grids) to one that is economy based aiming to deliver more abstract resources and services (such is the case in Clouds). As for Utility Computing, it is not a new paradigm of computing infrastructure; rather, it is a business model in which computing resources, such as computation and storage, are packaged as metered services similar to a physical public utility, such as electricity and public switched telephone network. Utility computing is typically implemented using other computing infrastructure (e.g. Grids) with additional accounting and monitoring services. A Cloud infrastructure can be utilized internally by a company or exposed to the public as utility computing.

See Figure 1 for an overview of the relationship between Clouds and other domains that it overlaps with. Web 2.0 covers almost the whole spectrum of service-oriented applications, where Cloud Computing lies at the large-scale side. Supercomputing and Cluster Computing have been more focused on traditional non-service applications. Grid Computing overlaps with all these fields where it is generally considered of lesser scale than supercomputers and Clouds.

**COMPUTE CLOUD Vs DATA CLOUD**
Traditional business model for software has been a one-time payment for unlimited use (usually on 1 computer) of the software. In a cloud-based business model, a customer will pay the provider on a consumption basis, very much like the utility companies charge for basic utilities such as electricity, gas, and water, and the model relies on economies of scale in order to drive prices down for users and profits up for providers.

Today, Amazon essentially provides a centralized Cloud consisting of Compute Cloud EC2 and Data Cloud S3. The former is charged based on per instance-hour consumed for each instance type and the later is charged by per GB-Month of storage used. In addition, data transfer is charged by TB / month data transfer, depending on the source and target of such transfer. The prospect of needing only a credit card to get on-demand access to 100,000+ processors in tens of data centers distributed throughout the world—resources that be applied to problems with massive, potentially distributed data, is exciting!

**ARCHITECTURE**
There are also multiple versions of definition for Cloud architecture, we define a four-layer architecture for Cloud Computing in comparison to the Grid architecture, composed of fabric, unified resource, platform, and application Layers. The fabric layer contains the raw hardware level resources, such as compute resources, storage resources, and network resources. The unified resource layer contains resources that have been abstracted/encapsulated (usually by virtualization) so that they can be exposed to upper layer and end users as integrated resources, for instance, a virtual computer/cluster, a logical file system, a database system, etc. The platform layer adds on a collection of specialized tools, middleware and services on top of the unified resources to provide a
development and/or deployment platform. For instance, a Web hosting environment, a scheduling service, etc. Finally, the application layer contains the applications that would run in the Clouds.

Clouds in general provide services at three different levels, although some providers can choose to expose services at more than one level.

**Infrastructure as a Service (IaaS)** provisions hardware, software, and equipments (mostly at the unified resource layer, but can also include part of the fabric layer) to deliver software application environments with a resource usage-based pricing model. Typical examples are Amazon EC2 (Elastic Cloud Computing) Service and S3 (Simple Storage Service) where compute and storage infrastructures are open to public access with a utility pricing model; Eucalyptus is an open source Cloud implementation that provides a compatible interface to Amazon’s EC2, and allows people to set up a Cloud infrastructure at premise and experiment prior to buying commercial services.

**Platform as a Service (PaaS)** offers a high-level integrated environment to build, test, and deploy custom applications. Generally, developers will need to accept some restrictions on the type of software they can write in exchange for built-in application scalability. An example is Google’s App Engine [28], which enables users to build Web applications on the same scalable systems that power Google applications.

**Software as a Service (SaaS)** delivers special-purpose software that is remotely accessible by consumers through the Internet with a usage-based pricing model. Salesforce is an industry leader in providing online CRM Services. Live Mesh from Microsoft allows files and folders to be shared and synchronized across multiple devices.

Although Clouds provide services at three different levels (IaaS, PaaS, and Saas), standards for interfaces to these different levels still remain to be defined. This leads to interoperability problems between today’s Clouds, and there is little business incentives for Cloud providers to invest additional resources in defining and implementing new interfaces. As Clouds mature, and more sophisticated applications and services emerge that require the use of multiple Clouds, there will be growing incentives to adopt standard interfaces that facilitate interoperability in order to capture emerging and growing markets in a saturated Cloud market.

**DYNAMIC MEMORY MANAGEMENT**

As we know that the most powerful model of application development is Object Oriented Programming Model to manage complex projects.

When a program is executed and suppose RAM is free to accept it then it hold storage in RAM as a process where size of process is greater than program size due to different storage region it required for smooth execution. Text region contains whole executable code, data section contains global variables, stack contains temporary data (such as function parameters, return addresses, and local variables) and heap is responsible to dynamically allocate memory during process run time.

```
{
    // ...
    Box b1 = new Box(3,4,5);
    double volume = b1->volume();
    // ...
    delete b1;
}
```

Figure 2: Cloud Architecture

Figure 3: Process in memory
In above code new and delete operators are used for allocation and cleanup respectively. If a programmer is forget to reclaim object storage from heap using delete operator then program is no longer to use object storage during its whole execution. This side effect is called Memory Leak in programming literature.

One solution to above problem is garbage collector software which reclaims memory automatically as it find that particular objects are no longer in use. JVM and .Net Frameworks have their integrated garbage collector. Garbage Collector is an overhead for program execution but it ensures that there is no Memory Leak.

TRADITIONAL GC

Traditional GC implementations use a stop-the-world approach to recovering heap memory. An application runs until the heap is exhausted of free memory, at which point the GC stops all application code, performs a garbage collect, and then lets the application continue.

Figure 4 illustrates traditional STW pauses for GC activity that are typically unpredictable in both frequency and duration. Traditional GC is nondeterministic because of the amount of effort required to recover memory depends on the total amount and size of objects that the application uses, the interconnections between these objects, and the level of effort required to free enough heap memory to satisfy future allocations.

The sweep phase is responsible for examining the heap after marking has completed and reclaiming the dead objects' storage back into the free store for heap, making that storage available for allocation. As with the mark phase, the cost of sweeping dead objects back into the free memory pool can't be completely predicted. Although the number and size of live objects in the system can be derived from the mark phase, both their position within the heap and their suitability for the free memory pool can require an unpredictable level of effort to analyze.

TRADITIONAL GC SUITABILITY FOR RT APP

RT applications must be able to respond to real-world stimuli within deterministic time intervals. A traditional GC can't meet this requirement because the application must halt for the GC to reclaim any unused memory. The time taken for reclamation is unbounded and subject to fluctuations. Furthermore, the time when the GC will interrupt the application is traditionally unpredictable. The time during which the application is halted is referred to as pause time because application progress is paused for the GC to reclaim free space. Low pause times are a requirement for RT applications because they usually represent the upper timing bound for application responsiveness.

REAL TIME SYSTEMS & GARBAGE COLLECTION

RT application development distinguishes itself from general-purpose application development by imposing time restrictions on parts of the runtime behavior. When hard RT systems, such as heart monitors or defense systems, miss these deadlines, it’s considered a catastrophic failure of the entire system. In soft RT systems, missed deadlines can have adverse effects -- such as a GUI not displaying all results of a stream it’s monitoring -- but don't constitute a system failure.

In Java applications, the Java Virtual Machine (JVM) is responsible for optimizing the runtime behavior, managing the object heap, and interfacing with the operating system and hardware. Although this management layer between the language and the platform eases software development, it introduces a certain amount of overhead into programs. One such area is GC, which typically causes nondeterministic pauses in the application. Both the frequency and length of the pauses are unpredictable, making the Java language...
traditionally unsuitable for RT application development. Some existing solutions based on the Real-time Specification for Java (RTSJ) let developers side step Java technology’s nondeterministic aspects but require them to change their existing programming model.

Metronome is a deterministic garbage collector that offers bounded low pause times and specified application utilization for standard Java applications. The reduced bounded pause times result from an incremental approach to collection and careful engineering decisions that include fundamental changes to the VM. Utilization is the percentage of time in a particular time window that the application is permitted to run, with the remainder being devoted to GC. Metronome lets users specify the level of utilization an application receives. Combined with the RTSJ, Metronome enables developers to build software that is both deterministic with low pause times and pause-free when timing windows are critically small. This article explains the limitations of traditional GC for RT applications, details Metronome’s approach, and presents tools and guidance for developing hard RT applications with Metronome.

**METRONOME GC**

Metronome’s approach is to divide the time that consumes GC cycles into a series of increments called quanta. To accomplish this, each phase is designed to accomplish its total work in a series of discrete steps, allowing the collector to:

1. Preempt the application for very short deterministic periods.
2. Make forward progress in the collection.
3. Let the application resume.

This sequence is in contrast to the traditional model where the application is halted at unpredictable points, the GC runs to completion for some unbounded period of time, and GC then quiesces to let the application resume.

Although splitting the STW GC cycle into short bounded pauses helps reduce GC’s impact, this isn’t sufficient for RT applications. For RT applications to meet their deadlines, a sufficient portion of any given time period must be devoted to the application; otherwise, the requirements are violated and the application fails. For example, take a scenario where GC pauses are bounded at 1 millisecond: If the application is allowed to run for only 0.1 millisecond between every 1-millisecond GC pause, then little progress will be made, and even marginally complex RT systems will likely fail because they lack time to progress. In effect, short pause times that are sufficiently close together are no different from a full STW GC.

Figure 5 illustrates a scenario where the GC runs for the majority of the time yet still preserves 1-millisecond pause times:

![Figure 5: Short pause times but little application time](image)

**UTILIZATION**

A different measure is required that, in addition to bounded pause times, provides a level of determinism for the percentages of time allotted to both the application and GC. We define application utilization as the percentage of time allotted to an application in a given window of time continuously sliding over the application’s complete run. Metronome guarantees that a percentage of processing time is dedicated to the application. Use of the remaining time is at the GC’s discretion: it can be allotted either to the application or to the GC. Short pause times allow for finer-grained utilization guarantees than a traditional collector. As the time interval used for measuring utilization approaches zero, an application’s expected utilization is either 0% or 100% because the measurement is below the GC quantum size. The guarantee for utilization is made strictly on measurements the size of the sliding window. Metronome uses quanta of 500 microseconds in length over a 10-millisecond window and has a default utilization target of 70%.

Figure 6 illustrates a GC cycle divided into multiple 500-microsecond time slices preserving 70% utilization over a 10-millisecond window:

![Figure 6: Sliding window utilization](image)

Although splitting the STW GC cycle into short bounded pauses helps reduce GC’s impact, this isn’t sufficient for RT applications. For RT applications to meet their deadlines, a sufficient portion of any given time period must be devoted to the application; otherwise, the requirements are violated and the application fails. For example, take a scenario where GC pauses are bounded at 1 millisecond: If the application is allowed to run for only 0.1 millisecond between every 1-millisecond GC pause, then little progress will be made, and even marginally complex RT systems will likely fail because they lack time to progress. In effect, short pause times that are sufficiently close together are no different from a full STW GC.

Figure 5 illustrates a scenario where the GC runs for the majority of the time yet still preserves 1-millisecond pause times:

![Figure 5: Short pause times but little application time](image)
quantum, even if the target utilization would be preserved with back-to-back GC quanta. This ensures the application pause times are limited to the length of 1 quantum. However, if target utilization is specified to be below 50%, some instances of back-to-back GC quanta will occur to allow the GC to keep up with allocation.

Figures 7 and 8 illustrate a typical application-utilization scenario. In Figure 7, the region where utilization drops to 70% represents the region of an ongoing GC cycle. Note that when the GC is inactive, application utilization is 100%.

![Figure 7: Overall utilization](image)

**Figure 7: Overall utilization**

Section A of Figure 8 is a staircase graph where the descending portions correspond to GC quanta and the flat portions correspond to application quanta. The staircase demonstrates the GC respecting low pause times by interleaving with the application, producing a step-like descent toward the target utilization. Section B consists of application activity only to preserve utilization targets across all sliding windows. It's common to see a utilization pattern showing GC activity only at the beginning of the pattern. This occurs because the GC runs whenever it is allowed to (preserving pause times and utilization), and this usually means it exhausts its allotted time at the beginning of the pattern and allows the application to recover for the remainder of the time window. Section C illustrates GC activity when utilization is near the target utilization. Ascending portions represent application quanta, and descending portions are GC quanta. The sawtooth nature of this section is again because of the interleaving of the GC and application to preserve low pause times. Section D represents the portion after which the GC cycle has completed. This section's ascending nature illustrates the fact that the GC is no longer running and the application will regain 100% utilization.

**RUNNING AN APPLICATION WITH METRONOME**

Metronome is designed to provide RT behavior to existing applications. No user code modification should be required. Desired heap size and target utilization must be tuned to the application so target utilization maintains the desired application throughput while letting the GC keep up with allocation. Users should run their applications at the heaviest load they want to sustain to ensure RT characteristics are preserved and application throughput is sufficient. This article's Tuning Metronome section explains what you can do if throughput or utilization is insufficient. In certain situations, Metronome's short pause-time guarantees are insufficient for an application's RT characteristics. For these cases, you can use the RTSJ to avoid GC-incurred pause times.

**TUNING METRONOME**

It's important to understand the correlation between heap size and application utilization. Although high target utilization is desirable for optimal application throughput, the GC must be able to keep up with the application allocation rate. If both the target utilization and allocation rate are high, the application can run out of memory, forcing the GC to run continuously and dropping the utilization to 0% in most cases. This degradation introduces large pause times often unacceptable for RT applications. If this scenario is encountered, a choice must be made to decrease the target utilization to allow for more GC time, increase the heap size to allow for more allocations, or a combination of both. Some situations might not have the memory required to sustain a certain utilization target, so decreasing the target utilization at a performance cost is the only option.

Figure 9 illustrates a typical trade-off between heap size and application utilization. A higher utilization percentage requires a larger heap because the GC isn't allowed to run as much as a lower utilization would allow.

The relationship between utilization and heap size is highly application dependent, and striking an appropriate balance requires iterative experimentation with the application and VM parameters.
The relationship between utilization and heap size is highly application dependent, and striking an appropriate balance requires iterative experimentation with the application and VM parameters.

THE REAL TIME SPECIFICATION FOR JAVA
The RTSJ is a "specification for additions to the Java platform to enable Java programs to be used for real-time applications." Metronome must be aware of certain aspects of the RTSJ -- in particular, RealtimeThreads (RT threads), NoHeapRealtimeThreads (NHRTs), and immortal memory. RT threads are Java threads that, among other characteristics, run at a higher priority than regular Java threads. NHRTs are RT threads that can't contain references to heap objects. In other words, NHRT-accessible objects can't refer to objects subject to GC. In exchange for this compromise, the GC won't impede the scheduling of NHRTs, even during a GC cycle. This means NHRTs won't incur any pause times. Immortal memory provides a memory space that's not subject to GC; this means NHRTs are allowed to refer to immortal objects.

BASIC NEEDS OF ROBOTICS
As we all are aware of intelligent machine like robot. Scientists are trying best to make robot intelligent like humans. Robot is a mechanical machine which is controlled by microcontroller or microprocessor. Intelligence of robot depends on quality of processor, sensors and programming algorithms. Also duration to work continuity depends on battery capacity while discharging depends on weight, amount of movement, amount of processing, complexity of electronic cards, etc. As level of intelligence increases the cost of robot may increase exponentially like Surgical Robot cost is $1 Million.

CONCLUSION
If a Real Time Cloud is available then for intelligent machines like robots the complex processing may be done on RTC via request and response model. The complex processing is more desirable as level of intelligence increases in robots towards humans. As complex processing is centralized via RTC therefore cost and complexity of electronic cards will reduced significantly for robot. Specially, when large number of robots working on disaster sites where humans reach is not possible at all. RTC based robots will be able to work for longer time due to low power consumption as there is no complex processing in robots itself.

FUTURE SCOPE
The proposed model is not only good for intelligent robots but also for hospitals to cure their critical patients faster as complex test processing may be possible via RTC which is not possible at hospital premises.

For the success of proposed model we need to make cloud more faster in terms of response, fast wired/wireless internet connectivity, R&D of new internet protocols and new programming frameworks with RTGC for real time response on cloud, etc.

REFERENCES