A Framework for Opportunistic Cluster Computing using JavaSpaces[1]

Jyoti Batheja and Manish Parashar

Electrical and Computer Engineering, Rutgers University
94 Brett Road, Piscataway, NJ 08854
{jbatheja, parashar}@caip.rutgers.edu

Abstract. Heterogeneous networked clusters are being increasingly used as platforms for resource-intensive parallel and distributed applications. The fundamental underlying idea is to provide large amounts of processing capacity over extended periods of time by harnessing the idle and available resources on the network in an *opportunistic* manner. In this paper we present the design, implementation and evaluation of a framework that uses JavaSpaces to support this type of opportunistic adaptive parallel/distributed computing over networked clusters in a non-intrusive manner. The framework targets applications exhibiting coarse-grained parallelism and has three key features: (1) portability across heterogeneous platforms, (2) minimal configuration overheads for participating nodes, and (3) automated system state monitoring (using SNMP) to ensure non-intrusive behavior. Experimental results presented in this paper demonstrate that for applications exhibiting coarse grained parallelism, the opportunistic parallel computing framework can provide performance gains. Furthermore, the results indicate that monitoring and reacting to current system state minimizes intrusiveness.

1 Introduction

This paper presents the design, implementation and evaluation of a framework that uses JavaSpaces [1] to aggregate networked computing resources, and non-intrusively exploits idle resources for parallel/distributed computing. Traditional High Performance Computing (HPC) is based on massively parallel processors, supercomputers or high-end workstation clusters connected by high-speed networks. These resources are relatively expensive, and are dedicated to specialized parallel and distributed applications. Exploiting available idle resources in a networked system can provide a more cost effective alternative for certain applications. However, there are a number of challenges that must be addressed before such opportunistic adaptive cluster computing can be a truly viable option. These include: 1) **Heterogeneity:** Cluster environments are typically heterogeneous in the type of

[1] The research presented in this paper is based upon work supported by the National Science Foundation under Grant Number ACI 9984357 (CAREERS) awarded to Manish Parashar.

resources, the configurations and capabilities of these resources, and the available software, services and tools on the systems. This heterogeneity must be hidden from the application and addressed in the seamless manner, so that the application can uniformly exploit available parallelism. 2) **Intrusiveness:** Inclusion of the framework must minimize modifications to any existing legacy code or standard practices. Furthermore, a local user should not be able to perceive that local resources are being stolen for foreign computations. 3) **System configuration and management overhead:** Incorporating a new resource into the cycle stealing resource cluster may require system configuration and software installation. These modifications and overheads must be minimized so that the cluster can be expanded on the fly to utilize all available resources. 4) **Adaptability to system and network dynamics:** The availability and state of system and network resources in a cluster can be unpredictable and highly dynamic. These dynamics must be handled to ensure reliable application execution. 5) **Security and privacy:** Secure and safe access to resources in the cluster must be guaranteed so as to provide assurance to the users making their systems available for external computations. Policies must be defined and enforced to ensure that external application tasks adhere to the limits and restrictions set on resource/data access and utilization.

Recent advances in opportunistic cluster computing have followed two approaches, *Job level parallelism* and *Adaptive parallelism*[2]. In the job level parallelism approach, entire application jobs are allocated to available idle resources for computation, and are migrated across resources as resources become unavailable. The Condor [10] system supports cluster-based job level parallelism. In the adaptive parallelism approach, the available processors are treated as part of a dynamic resource pool. Each processor in the pool aggressively competes for application tasks. This approach targets applications that can be decomposed into independent tasks. Adaptive computing techniques can be *cluster based* or *web based*. *Cluster based* systems exploit available resources within a local networked cluster. *Web based* approach extends this model to resources over the Internet. Systems supporting adaptive parallelism include cluster-based systems such Piranha [11][12], Atlas [6], and ObjectSpace/Anaconda [9], and web-based systems such as Charlotte [4][5], Javelin [8], and ParaWeb [7].

This paper presents the design, implementation and evaluation of a framework for adaptive and opportunistic cluster computing based on JavaSpaces that address the issues outlined above. The framework has three key features: (1) portability across heterogeneous platforms, (2) minimal configuration overheads and runtime class loading at participating nodes, and (3) automated system state monitoring (using SNMP [2][3]) to ensure non-intrusive behavior.

The rest of this paper is organized as follows. Section 2 describes the architecture and operation of the proposed framework. Section 3 presents an experimental evaluation of the framework. Section 4 presents our conclusions and outlines current and future work.

---

[2] To best of our knowledge, the term “adaptive parallelism” was coined by the Piranha project [12].

2 A Framework for Opportunistic Parallel Computing on Clusters

The framework presented in this paper employs JavaSpaces to facilitate master-worker parallel computing on networked clusters. JavaSpaces is a Java implementation of a tuple-space system [13], and is provided as a Jini service [18]. JavaSpaces technology provides a programming model that views applications as a collection of processes cooperating via the flow of objects into and out of one or more spaces. A space is a shared, network accessible repository for objects [14]. In the presented framework, parallel workload is distributed across the worker nodes using the bag of task model with the master producing independent application tasks into the space, and the worker consuming these tasks and computing on them. Results are returned to the space. This model offers two key advantages. (1) The model is naturally load-balanced. Load distribution in this model is worker driver. As longs as there work to be done, and the worker is available to do work, it can keep busy. (2) The model is naturally scalable. Since the tasks are relatively independent, as longs as there are a sufficient number of task, adding workers improves performance.

The framework and underlying parallel computing model supports applications that are sufficiently complex and require parallel computing, that are divisible into relatively coarse-grained subtasks that can be solved independently, and where the subtasks have small input/output sizes.

2.2 Framework Architecture

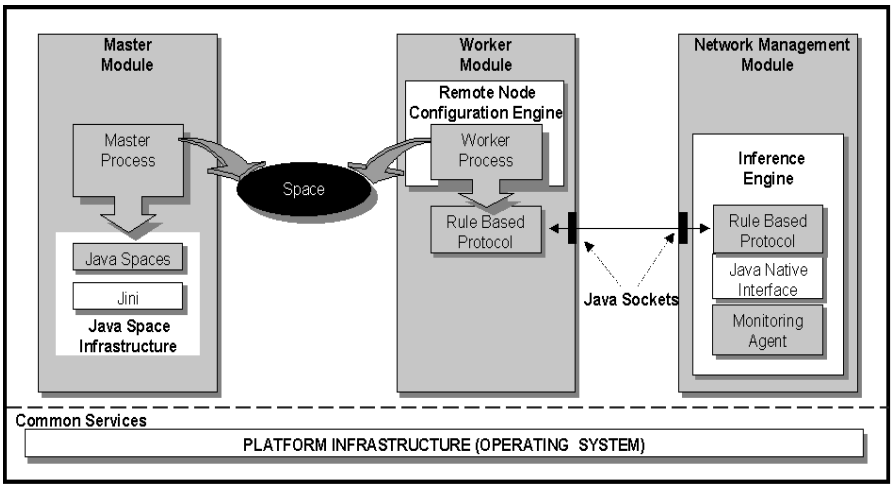

Fig. 1. Framework architecture

A schematic overview of the framework architecture is shown in Fig. 1. It consists of 3 key components: the Client-side (Master) components, the Server-side (Worker) components and the Network Management Module.

Master Module: The Master component defines the problem domain for a given application. The application domain is broken down into sub tasks that are JavaSpace enabled.[3] The master also contains the JavaSpace and registers it as a Jini service. It relies on Jini for remote lookup during the discovery phase. The JavaSpace is used to handles all communication issues.

Worker Module: The worker component provides the solution content for the application domain. In an effort to minimize the overheads of deploying worker code, we have implemented a remote node configuration mechanism that facilitates remote loading of the worker implementation classes at runtime.

Network Management Module: In order to exploit idle resources while maintaining non-intrusiveness at the remote nodes, it is critical that the framework monitors the state of the worker nodes, and uses this state information to drive the scheduling of tasks on workers. The *Network Management Module* performs this task. It monitors the state of registered workers and uses defined policies to decide on the workers availability. The policies are maintained by the Inference Engine component and enforced using the Rule Base Protocol.

2.2 Implementation and Operation

The framework implements the master-worker pattern with JavaSpaces as the backbone. The overall operation of the framework consists of three potentially overlapping phases, viz. task-planning, compute, and result-aggregation. During the *task-planning phase*, the master process first decomposes the application problem into sub tasks. It then iterates through the application tasks, creates a task entry for each task, and writes the tasks entry into the JavaSpace. During the *compute phase*, the worker process collects these tasks from the JavaSpace. Matchmaking in JavaSpaces is achieved by identifying each task object by a unique ID and the space where it resides. If a matching task object is not available immediately, the worker process waits until one arrives. The worker classes are downloaded at runtime using the Remote Node Configuration Engine. Remote node configuration is explained in section 2.2.1. Results obtained from executing the computations are put back into the space. During the compute-phase, if the resource utilization on the worker nodes becomes intolerable the rule base protocol sends a stop/pause signal to the worker process. On receiving the signal, the worker process completes the execution of the current task and returns its results into space. It then enters the stop/pause state and does not accept tasks until it receives a start/resume signal. During the *result aggregation* phase, the master process removes results written into the space by the workers, and aggregates them into the final solution.

---

[3] JavaSpace required the Objects being passed across the Space to be in a Serializable format. In order to transfer an entry to or from a remote space, the proxy to the remote space implementation first serializes the fields and then transmits it into the space.

2.2.1 Remote Node Configuration

The required classes for remote configuration of the worker nodes are easily downloadable from the web server residing at the master in the form of executable jar files. The application implementation classes are loaded at runtime from within the configuration classes, and the appropriate method to start the worker application thread is invoked. Our modification of the network launcher [15] provides mechanisms to intercept calls from the inference engine (the network management module) and interpret them as signals to the executing worker code. This interaction is used to enable the worker to react to system state as explained below.

2.2.2 Rule Base Protocol

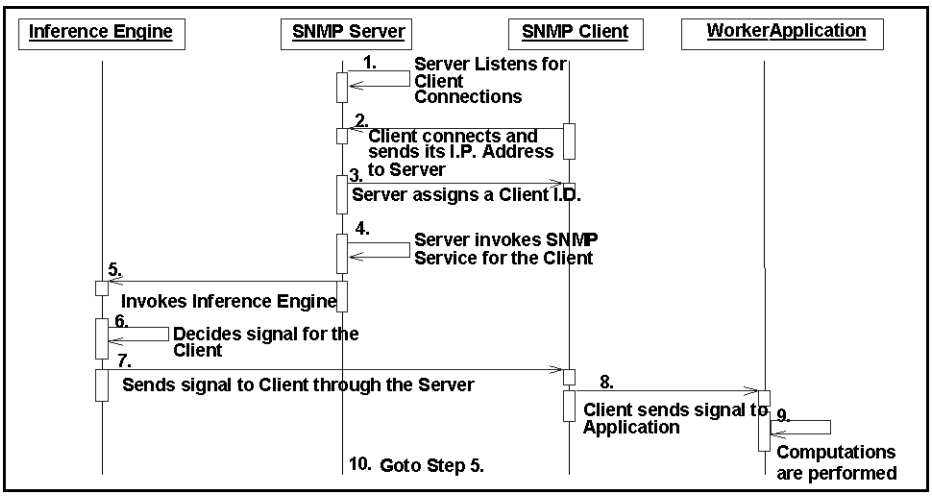

Fig. 2. Sequence diagram for the rule base protocol

The rule base protocol defines the interaction between the network management module and the worker module (see Fig. 2) to enable the worker to react to changes in its system state. It operation is as follows:

The SNMP client, which is part of the worker module, initiates the workers participation in the parallel computation by registering with the SNMP server at the network management module. The inference engine, also at the network management module, maintains a list of registered workers. It assigns a unique ID to the new worker and adds its IP address to the list. The SNMP server then continues to monitor the state of workers in its list.

The SNMP parameter monitored is the average worker CPU utilization. As these values are returned they are added to the respective entry in the server list. Based on this return value and programmed threshold ranges, the inference engine makes a decision on the workers current availability status and passes an appropriate signal back to the worker. Threshold values are based on heuristics. The rule base currently

defines 4 types of signals in response to the varying load conditions at a worker; viz. *start, stop, pause* and *resume*.

Start: This signal is sent to the worker nodes to signify that the worker node is now idle and can start the parallel processing job. The threshold average CPU load values for this state are in the range of 0% - 25%. On receiving this signal, the worker initiates a new runtime process that loads the application worker classes and starts work execution.

Stop: This signal is sent to the worker nodes to indicate that the worker node can no longer be used for computations. This may be due to a sustained increase in CPU load caused by a higher priority (possibly interactive) job being executed. The cutoff threshold value for the stop state is an average CPU utilization greater than 50%. On receiving the stop signal, the executing worker thread is interrupted and shutdown/cleanup mechanisms are initiated. The shutdown mechanism ensures that the currently executing task completes and its results are written into the space. After cleanup the worker thread is killed and control returns to the parent process.

Pause: This signal is sent to the worker nodes to indicate that the worker node is experiencing increased average CPU loads. However, the load increase might be transient and node could be reused for computation in the near future. Hence it should temporarily not be used for computation. Threshold values for the pause state are in the range of 25% - 50%. Upon receiving this signal, the worker backs off, but unlike the stop state the back off is temporary, i.e. until it get the resume signal. This minimizes worker initialization and class loading overheads for transient load fluctuations. As in the stop state, the pause goes into effect only after the worker writes the results of the currently executing task into the space.

Resume: This signal is sent to the worker nodes, while paused, to indicate that the worker node is once again available for computation. This signal is triggered when the average CPU load falls below 25%. Upon receiving this signal the worker process once again retrieves tasks from the space and computes them.

3 Framework Evaluation

We evaluated the JavaSpaces-based opportunistic cluster-computing framework with a real world financial application that uses Monte Carlo (MC) simulation for Option Pricing. An option is a derivative, that is, its pricing value is derived from something else. Complications such as varying interest rates and complex contingencies can prohibit analytical computation of options and other derivative prices. Monte Carlo (MC) simulation [17] using statistical properties of assumed random sequences is an established tool for pricing derivative securities. An option is defined by the underlying security, the option type (call or put), the strike price, interest rate, volatality and the expiration date. These financial terms are explained in greater depth in [16]. The main MC simulation based on the input parameters is the core parallel computation in our experiments. Input parameters may be defined using a GUI as provided in our implementation. The simulation domain is divided into tasks

of size 100 each and MC simulations are performed in parallel on these tasks. High and low pricing estimates are obtained over a wide range of simulations.

3.1 Scalability Analysis

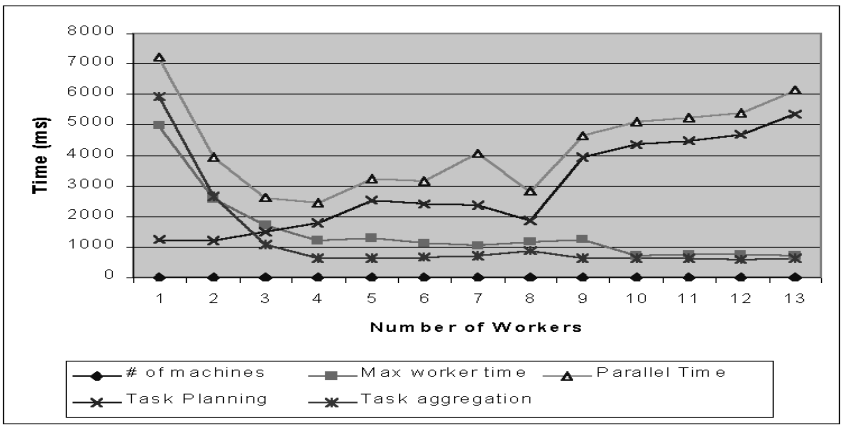

Fig. 3. Application scalability

This experiment measures the overall scalability of the application and the framework. Results for this experiment are plotted in Fig. 3. As shown in the figure an initial speedup is obtained as the number of workers is increased. During this part of the curve the total parallel time closely follows the maximum worker time. As the number of workers increases the model spreads the total tasks more evenly across the available workers. Hence the maximum (Max) worker time evens out as the number of workers increases. However, after a point we notice that the total parallel time is dominated by the Task Planning time. That is the workers are able to complete the assigned task and return it to the space much before the master gets a chance to plan a new task and put it into the space. Hence the workers remain starved until the task is made available. As a result the scalability deteriorates. This indicates that the framework favors coarse-grained tasks that are compute intensive. As expected the task aggregation curve closely follows the maximum worker time.

3.2 Adaptation Protocol Analysis

In this experiment, we provide a time analysis to illustrate the overhead involved in signaling worker nodes and adapting to their current CPU load. As a part of the experimental setup, we built two sets of load simulators: load simulator 1 was designed to raise the CPU usage level on the worker to 30% to 50% utilization. The second load simulator (load simulator 2) raised the CPU utilization of the worker machines to 100%. Fig. 4(a) and Fig. 4(b) depict the worker behavior under the

simulation conditions. Fig. 4(a) captures the CPU usage history on the worker host throughout the run. We identify the peaks where the worker reacts to the signals sent. The first peak at 80% CPU usage occurs when the worker is started. This sudden load increase is attributed to the remote loading of application classes at the worker. Next, load simulator 2 is started which sends the CPU usage to 100%. This causes a Stop signal to be sent to the worker node. The load simulator 2 is then stopped and load simulator 1 is started which raises the CPU load to 46%. As seen in Fig. 4(b) the worker reaction times to the signal is minimal in all cases. Furthermore, the large overhead associated with remote class loading is avoided in the case of transient load increase at the node using the pause/resume states.

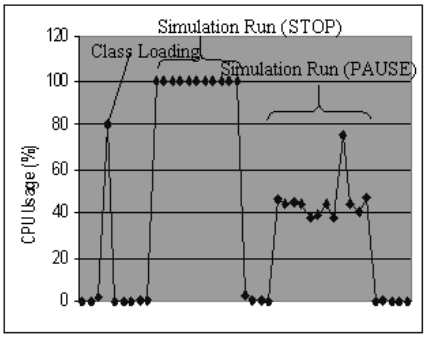

Fig. 4(a). Worker CPU usage

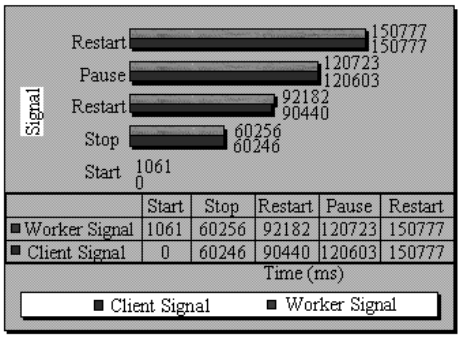

Fig. 4(b). Worker reaction times

Simulation signal triggers: Start - Stop - Restart - Pause - Resume

3.3 Dynamic behavior patterns under varying load conditions

This experiment consists of three runs: In the first run none of the workers were loaded. In the second and third runs, the load simulator 2 was run to simulate high CPU loads on 3 and 6 workers respectively. As seen in Fig. 5(a), as the number of worker hosts being loaded increases, the total parallel computation time increases. The computational tasks that would have been executed normally at a worker are now off loaded and picked up by other executing workers. The task planning and aggregation times also increase since the master has to wait for the worker with the maximum number of tasks to return its results back into the space. The maximum master overhead and the maximum worker time remains the same across all three runs as expected. Fig. 5(b) illustrates how task scheduling adapts to load current load conditions. It shows that the number of task executed by each worker depends on its current load. Loaded workers execute fewer tasks causing the available workers to execute larger number of tasks.

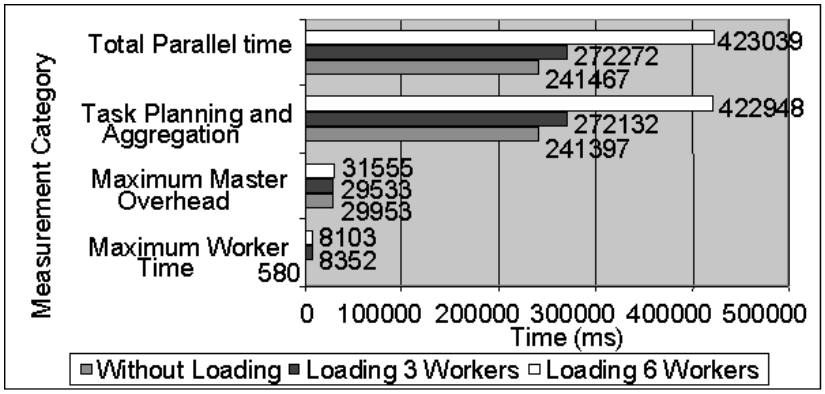

Fig. 5(a). Execution time measurements for 12 workers

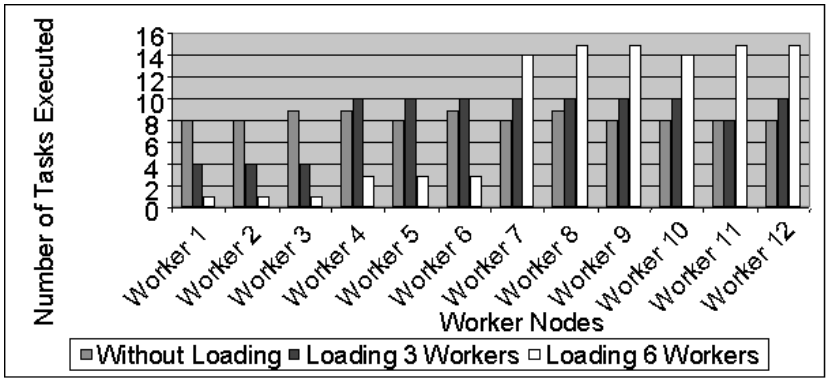

Fig. 5(b). Number of tasks executed per worker for 12 workers

4 Conclusions

This paper presented the design, implementation and evaluation of a framework for opportunistic parallel computing on networked clusters using JavaSpace. It provides support for global deployment of application code and remote configuration management of worker nodes, and uses an SNMP system state monitor to ensure non-intrusiveness. The experimental evaluation, using an option pricing application, shows that the framework provides good scalability for coarse-grained tasks. Furthermore, using the system state monitor and triggering heuristics the framework can support adaptive parallelism and minimize intrusiveness. The results also show that the signaling times between the worker and network management modules and the overheads for adaptation to cluster dynamics is insignificant. We are currently investigating ways to reduce the overheads during task planning and allocation phases. As future work, we envision incorporating a distributed JavaSpaces model to

avoid a single point of resource contention or failure. The Jini community is also investigating this area.

References

1.Sun Microsystems. Javaspaces, www.javasoft.com/products/javaspaces/specs/ (1998)
2. SNMP Documentation, http://www.snmpinfo.com
3. James Murray, Windows NT SNMP, O'Reilly Publications (January 1998)
4. A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In Proceedings of the 9th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS), (September 1996) 181-188
5. A. Baratloo, M. Karaul, H. Karl, Zvi M. Kedem. An Infrastructure for Network Computing with Java Applets. Concurrency: Practice and Experience, Vol. 10, (September 1998) 1029-1041
6. J. Baldeschwieler, R. Blumofe, and E. Brewer. ATLAS: An Infrastructure for Global Computing. In Proceedings of the 7th ACM SIGOPS European Workshop: Systems support for Worldwide Applications (September 1996) 165-172
7. T. Brecht, H. Sandhu, J. Talbott, and M. Shan. ParaWeb: Towards world-wide supercomputing. In Proceedings of the 7th ACM SIGOPS European Workshop (September 1996) 181-188
8. B. Christiansen, P. Cappello, M. Ionescu, M. Neary, K. Schauser, and D. Wu. Javelin: Internet-based parallel computing using Java. Concurrency: Practice and Experience, Vol. 9 (November 1997) 1139-1160
9. P. Ledru. Adaptive Parallelism: An Early Experiment with Java™ Remote Method Invocation. Technical Report, CS Department, University of Alabama (1997)
10.M. Lizkow, M. Livny, and M. Mukta. Condor: A hunter of idle workstations. In Proceedings of the 8th International conference on Distributed Computing Systems (June1998) 104-111
11. N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive parallelism and Piranha. IEEE Computer, Vol. 28, No. 1 (January 1995) 40-49
12. D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary Experience with Piranha. In Proceedings of the 6th ACM International Conference on Supercomputing (July 1992) 417-427
13.The Linda Group. http://www.cs.yale.edu/HTML/YALE/CS/Linda/linda.html
14. E. Freeman, S. Hupfer, K. Arnold. JavaSpaces Principles, Patterns, and Practice. Addison Wesley (June 1999)
15.M. Noble. Tonic: A Java TupleSpaces Benchmark Project. http://hea-www.harvard.edu/~mnoble/tonic/doc/
16. Glossary of Financial terms http://www.centrex.com/terms.html
17.Broadie and Glasserman MC algorithm for Option Pricing http://www.puc-rio.br/marco.ind/monte-carlo.html
18. W. Keith Edwards. Core Jini, Addison Wesley (October 2000).