Extending Peer-to-Peer Computing Infrastructures for Computing Cycle Sharing

Ali Raza Butt

Y. Charlie Hu

School of Electrical and Computer Engineering
1285 Electrical Engineering Building
Purdue University
West Lafayette, IN 47907-1285

Contents

Abstract

Compute–cycle sharing across machines can facilitate fast solutions of complex scientific problems. Therefore, software for distribution of problem sets to a large number of nodes is becoming increasingly important. Structured peer–to–peer (p2p) overlay networks, which implement scalable and fault-tolerant distributed hashing tables primarily for data sharing, has also seen a boost. This report proposes a novel approach for bringing the two worlds of cycle–sharing and data sharing together to yield a decentralized distributed computer that is more scalable and robust than centralized computation distribution schemes. This work is unique as it shows that p2p systems are not limited to data sharing applications, they can also form a suitable substrate for computing cycle sharing. Its access is not limited to distribution managers; instead all users of the system can access the resources available. The approach leverages p2p overlays in conjunction with the programming model laid down in the Java Application Isolation API. The issues of computation migration, integrity, routing, and failure resiliency with regard to the proposed scheme are also addressed.

Key Phrases: Compute cycle sharing, peer–to–peer computing, distributed computing.

1. Introduction

Compute–cycle sharing for complex and large scientific problems is a long studied topic in distributed computing. Many cycle–sharing setups such as SETI@Home [18], Distributed.Net [5], and Entropia [6] have shown that complex scientific problems can be quickly solved by distributing them to thousands of clients. There are several issues associated with this setup: a) The distribution manager becomes a single point of failure as well as a performance bottleneck, as all communication is routed through it; b) There is an absence of direct communication between the clients, which restricts the ways a problem can be distributed; c) The clients are unable to spawn sub–computations; d) Only the manager has access to use the shared resources; and e) There is an explicit allocation of clients, which limits the available resources, and does not provide any resiliency in choosing a client. In contrast to compute–cycle sharing setups, p2p systems consist of clients that are identical in capabilities and all clients can perform symmetric communications. There is natural replication among clients which provides resiliency and fault tolerance. Therefore, all clients can use the system for solving problems. However, p2p systems so far have aimed at sharing data, and there is no well–established compute–cycle sharing paradigm associated with the p2p systems. The key idea of this paper is to extend both the p2p infrastructure and the distributed programming model in order to support a robust distributed computer.

To leverage p2p infrastructures developed for sharing data to support distributed computations, some means need to be provided for transforming computations to some "images" that can then be treated as data. One way of achieving this on traditional Unix systems is by freezing the computation, checkpointing [22, 21, 14] the process image to a file

and then using structured p2p systems such as CAN [16], Chord [19], Pastry [17], and Tapestry [23] to distribute the checkpoint to remote nodes. The problem is complicated by the fact that checkpointing is machine dependent, whereas p2p clients are inherently heterogeneous. Even a machine–independent checkpointing approach as described in [15] is insufficient since an elaborate procedure is required on each node for migration and execution. Furthermore, checkpointing approaches usually require special user accounts on all the shared resources; an approach that has many problems [3]. For this reason, a new programming paradigm as described in JSR121 [8] is adopted for the presented scheme. With the help of this model, the standard p2p system can be extended to provide robust cycle sharing. The proposed scheme does not require creation of special user accounts on shared resources and is more practical.

The underlying problem of node selection can be solved using the routing mechanisms of structured p2p systems. The desire to be able to migrate the computation from one node to another dictates that the programming model should a) be easy to adapt applications to; b) be efficient, i.e. execution environment can be easily setup without unnecessarily loading the node; c) provide means for encapsulating whole computations as programming objects; d) support functions on the programming objects such as stopping and starting computations on demand; e) protect two objects from each other; and f) be machine independent so that computations can be executed on heterogeneous machines. Java is a good candidate for providing machine independence. However, setting up a Java Virtual Machine (JVM) for each migrated computation is expensive [2]. Therefore a new programming model is required.

The rest of the report is organized as follows. Chapter 2 presents the Java Application Isolation model. Chapter 3 describes a set of extensions to the isolate model for use in distributed systems. Chapter 4 describes a technique to achieve the computing cycle sharing using p2p systems. Finally, Chapter 5 presents a brief discussion of related work and gives concluding remarks.

2. The Java Application Isolation Model

The Java Application Isolation model is described in [8, 7]. It provides a programming paradigm based on isolated computation entities. An *isolate* is defined as a Java object that encapsulates a complete computation. In some respects, it is similar to a process in operating systems. It provides facilities that the JVM can use to start, suspend, resume and terminate the computation referred by the isolate. Each isolate can perform a computational task similar to a process. However, Figure 2.1 illustrates main differences between processes and isolates: Isolates are Java objects and multiple isolates can coexist in different portions of the same address space without interfering or even detecting each other. Address space sharing is not possible for processes. Threads, however, share an address space, but there is no protection between coexistent threads. In this regard, an isolate can be thought of a self–contained thread, that has more capability in terms of isolation.

Since one instance of JVM can only have a single address space [10], creation of a separate address space on a node implies spawning a separate JVM. This is an expensive process, which can be minimized by sharing a JVM. But, protection problems prohibit the use of threads for such sharing. Isolates, on the other hand, are guaranteed a safe address space sharing by type safety in the language. They can run safely in a single instance of a JVM and avoid all overheads associated with spawning a new JVM. This makes the Java Isolate API ideal for efficient computation migration in wide area networks.

Since the model is Java based, adapting applications to it is not overwhelming, and heterogeneity issues are solved automatically. Overall, the Java Application Isolation API fits all the requirements of the programming model for cycle sharing as listed in Chapter 1.

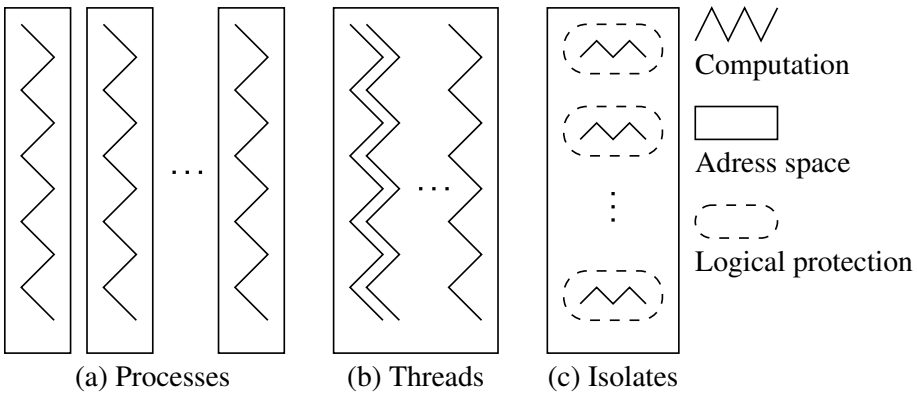

Figure 2.1. Differences between processes, threads and isolates. (a) For processes each computation is enclosed in its own address space. (b) For threads multiple computations can share an address space, but without protection. (c) For isolates each computation has its own logical protected subspace, guaranteed by safety in the language. Multiple isolates can co–exist in the same address space.

3. Extending the Isolation Model for Distributed Execution

For distributed execution, the Isolation Model alone is not sufficient. The following sections discuss the issues faced in extending the isolate model for use in distributed cycle sharing.

3.1 Execution Environment

In order to utilize the systems, the problem should be coded using the Application Isolation API, and submitted to the system. Upon receipt of the code, the execution environment starts the isolate using the start method, and then suspends it using the suspend method. This ensures uniformity by enabling clients to always execute an isolate via resume method. An appropriate node for the execution of the application is then selected as described in Chapter 4. Once the node is chosen, the isolate is migrated to that node (as explained in the next Section) for execution. If a node decides that it can no longer spare resources for the computation, the computation is suspended and the submitter is informed. The stopped computation can either be stored locally for continuation later, or migrated to another node. Since many different computations can run in the same address space, the node needs to set up only one JVM at startup. As long as resources are available, new isolates can be distributed in the shared address space.

3.2 Isolate Migration

A migration facility is required to enable isolates to move from one node to another. The problem of migration is somewhat simplified as an isolate is a Java object, and therefore can be serialized using the Java Serialization API [20].

To migrate an isolate, a node proceeds as follows. The suspend method is called on the target isolate to suspend the computation. The serialization API is then invoked to create a serial image of the isolate. This image is hashed and inserted into the system to be sent to a new node. Upon arrival at a node, the image is de–serialized and instantiated in the shared JVM. The resume method is used to transparently restart the computation, completing the migration of an isolate.

If processes were used instead of isolates, spawning new JVMs on the target nodes would become necessary. This is an expensive process. Alternatively, if threads were used to share a single JVM, unexpected problems could occur due to unexpected interference with already running threads. An involved analysis would be required to determine the safety of such interference. Isolates are free from these issues. They are safe to migrate to a node as long as it has resources to spare, and no extensive interference analysis is required due to safety guarantees in the language. Hence, isolate migration is more efficient than process migration and safer than thread migration.

The serialization and migration performed here is different than Java Remote Method Invocation (RMI). In RMI the caller process explicitly locates a target node and is aware of it throughout the life of the computation. In this case, the computation is completely handed off to a remote node without any knowledge about that node. Moreover, isolates are lightweight and efficient than RMI as discussed in [13].

3.3 Isolate Fork

The ability for an isolate to fork a new isolate is also provided. For this purpose a data structure called *neighbor set* is added to each isolate image. The *neighbor set* is defined as a set containing information about the parent and immediate children of a node. When a fork is required, an isolate creates a copy of itself, serializes the copy, attaches the information

about which data to process, adds itself as the parent node in the *neighbor set* of the image and then inserts it in the system to be migrated. A parent/child computation tree is formed. When an isolate arrives at a new node, the JVM retrieves the *neighbor set* and data information and execute the isolate similarly as done in migration. The child node then communicates to the parent and decides on what problem set to process. On completion of the computation, each node passes the result to its parent. In this way, computations can be tracked and results can be formed by traversing the parent tree.

3.4 Isolate Communication

The isolates can support a communication library called Links [13], which allows isolates in a shared address space to exchange information. In a distributed scenario, the basic communication facility has to be extended to provide information exchange between isolates running on distributed nodes.

Cross virtual machine communication can be set up by implementing the Links API over the `Sockets` API. In this way, the isolates can communicate between any two nodes independent of whether they run in the same address space or not. This approach is easier to implement, but poses challenges to the programmer as it exposes the fact that some isolates may be remote.

A **multi–node virtual machine** can be established by extending a single virtual machine across multiple nodes [1, 13]. Once again, the underlying communication between nodes is provided using the `Sockets` API. In this case, however, the location of isolates is not exposed to the programmer. This results in an easier programming model without special considerations for local and remote isolates. The multi–node virtual machine coupled with the proposed scheme for cycle sharing can result in a distributed virtual machine implementation whose underlying nodes are self-organized via a p2p infrastructure. Such a virtual machine is inherently scalable and fault tolerant.

4. Cycle Sharing in P2P Systems

It is proposed that available computing resources are organized into a peer-to-peer overlay network and the above isolate model is extended for distributed execution for compute cycle sharing among the users by using the p2p overlay. Structured p2p overlay networks such as CAN[16], Chord[19], Pastry[17], and Tapestry[23] effectively implement scalable and fault-tolerant *distributed hash tables* (DHTs), where each node in the network has a unique nodeId and each data item stored in the network has a unique key. The nodeIds and keys live in the same namespace, and each key is mapped to a unique node in the network. Thus DHTs allow *data* to be inserted without knowing where it will be stored and requests for data to be routed without requiring any knowledge of where the corresponding data items are stored.

The computing cycles in a p2p overlay of computing nodes can be shared as follows. When an isolate is submitted to the p2p system for execution, the initiating node creates a suspended isolate as described in Chapter 3. To migrate the suspended isolate or a forked isolate to another computing node, the code is first signed for integrity and protection from malicious nodes. The JVM on the initiating node then simply takes a hash of the signed code, and uses the underlying p2p system to route the code with the hashed key to a suitable node (child node). If the child node does not have free cycles, it can use a technique similar to "replica diversion" in PAST to divert the migrated isolate to some node in its leafset. If none of the nodes in its leafset have free cycles, it returns a negative acknowledgment to the parent node, and the parent node can use a technique similar to "file diversion" in PAST to generate a new key and repeat the migration process.

In the following sections, several issues related to data distribution, fault tolerance, and

security in the p2p–based cycle sharing infrastructure are discussed.

4.1 Data distribution

An important aspect of the problem distribution is the distribution of data. Two techniques can be used. Either the data can be packed with the computation and distributed with the isolate, or the clients can communicate with the data source for their data needs. If the data is packed with the isolate, a whole new paradigm will have to be set up to ensure that the problem set distribution is correct. However, if the isolates are only used for code distributions and data is retrieved by the clients only when they start computations, the time tested techniques of problem set distribution in distributed computation systems such as SETI@Home [18], Distributed.Net [5], and Entropia [6] can be leveraged. Therefore, in this scheme data propagation is orthogonal to code distribution.

4.2 Fault tolerance

An advantage of p2p system is that if a failure occurs, the underlying p2p system automatically finds an alternative node for re-execution, hence, providing fault tolerance. This is an important advantage over traditional ways, where failure of a client implies explicit reselection.

For reasons of failure resiliency and/or malicious node detection via comparison of results from multiple children, a node may create multiple similar computation branches by forking identical children. In case a parent has failed by the time a child completes, the child discards the results. There is no point of trying to send the results to a grand parent, as it may not be aware of how the data set was distributed among its various grand children, and without this knowledge any results from a grand child are meaningless.

4.3 Security

It is shown in [3] that active enforcement of security polices is required when sharing code from arbitrary users on shared resources. The problem is two–pronged, the code can

be malicious and tries to compromise the shared resource, or the resource can be malicious and tries to jeopardize the results of a computation. Since the scheme employs the Java Virtual Machine, which provides effective sandboxing [4], the host machine is protected from any the submitted code of malicious nature. On the other hand, computation replication and comparison can provide some protection against malicious hosts trying to affect a calculation. Furthermore, the isolate execution environment can be extended to check the integrity of the executable. If any attempt is made at modifying it, it can be detected and the execution is aborted. In fact, all that is needed to guarantee is that the execution environment itself is unmodified. As long as it is unmodified, the executed code is guaranteed to run as intended, because a malicious host can only reach it via the execution environment.

5. Related Work and Conclusions

The work done in the fields of both p2p systems and computing cycle sharing are related to the proposed approach. Structured p2p systems such as CAN [16], Chord [19], Pastry [17], and Tapestry [23] implement distributed hashing tables and can be used to self-organize the computing nodes and for selecting nodes for computation migration. On the computing cycle sharing front, Condor [11] is a example of a system that uses a large number of idle computers for solving complex scientific problems. However, it has been shown that using ordinary UNIX environment for resource sharing has security implications [12, 3]. Kazaa [9] is an example of a p2p system that packages an independent distributed client with its p2p data sharing software, which in essence is identical to standard distributed clients of systems such as SETI@Home [18], Distributed.Net [5], and Entropia [6]. Hence, kazaa remains prone to the problems discussed earlier.

The paper makes following contributions: a) A Java Application Isolation API based scheme for distributed computing cycle sharing on top of p2p networks is described; b) Necessary extensions are made to the isolates for providing robust execution environment, migration and fork facilities, and inter–node communication; c) The p2p routing infrastructure is leveraged for fault tolerant, secure compute cycle sharing; and d) All users are allowed to utilize the computation resources, and the use is not limited to system managers. No specialized permissions etc. are required.

Although the Java Application Isolation has been used as the programming model for the proposed system, the presented approach can work equally well with any programming paradigm that provides application isolation via type safety and computation suspend/resume facilities. The resulting system is robust and efficient, and can provide a huge base of resources for solving complex scientific problems.

Bibliography

[1] J. Andersson, S. Weber, E. Cecchet, C. Jensen, and V. Cahill. Kaffemik - A distributed JVM on a Single Address Space Architecture.

[2] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, San Diego, CA, October 2000.

[3] A. R. Butt, S. Adabala, R. J. Figueiredo, N. H. Kapadia, and J. A. B. Fortes. Fine–grain access control for securing shared resources in computational grids. In *Proceedings of IPDPS'02*, April 2002. Ft. Lauderdale, FL.

[4] D. Dean, E. W. Felten, and D. S. Wallach. Java security: from HotJava to Netscape and beyond. In *1996 IEEE Symposium on Security and Privacy, Oakland, California*, 1996.

[5] Distributed Computing Technologies Inc. http://www.distributed.net.

[6] Entropia Inc. http://www.entropia.com.

[7] Grzegorz Czajkowski. Application Isolation in the Java(tm) Virtual Machine. In *Proceedings of ACM OOPSLA'00, MN*, October 2000.

[8] Java Community Process. JSR 121 – Application Isolation API Specification. http://www.jcp.org/jsr/detail/121.jsp, 2001.

[9] Kazaa. http://www.kazaa.com.

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1997.

[11] M. J. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th ICDCS*, 1988.

[12] B. P. Miller, M. Christodorescu, R. Iverson, T. Kosar, A. Mirgorodskii, and F. Popovici. Playing inside the black box: Using dynamic instrumentation to create security holes. *Parallel Processing Letters*, 11(2,3):267–280, 2001.

[13] K. Palacz. Crusoe – A Cluster Java Virtual Machine. Technical report, Purdue University, August 2002. PhD Thesis Preliminary Examination.

[14] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *USENIX Winter 1995 Technical Conference*, January 1995. New Orleans, LI.

[15] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogenous architectures. In *Digest of Papers - 27th International Symposium on Fault–Tolerant Computing*, pages 58–67, June 1997. Seattle, Washington.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM*, August 2001.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, November 2001.

[18] SETI. Institute online http://www.seti-inst.edu/science/setiathome.html.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM*, San Diego, California, August 2001.

[20] Sun (TM). Object Serialization http://java.sun.com/j2se/1.3/docs/guide/serialization/.

[21] M. L. V.C. Zandy, B.P. Miller. Process Hijacking. In *Eighth International Symposium on High Performance Distributed Computing (HPDC '99)*, pages 177–184, August 1999. Redendo Beach, CA.

[22] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. M. R. Kintala. Checkpointing and its applications. In *25th International Symposium on Fault-Tolerant Computing*, pages 22–31, June 1995. Pasadena, CA.

[23] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.