Java Replicated Objects

XIAN KE, HUBERT PHAM, SIDDHARTHA SEN and ADRIAN SOLIS
Massachusetts Institute of Technology

The ability to replicate whole objects across a network of machines, while maintaining data consistency, is a common requirement shared by many distributed applications. Existing systems either do not support object data consistency across copies existing on multiple nodes, or they suffer from a single point of failure. We present an application framework, Java Replicated Objects (JRO), that addresses these issues. Java Replicated Objects distributes copies of an object to different nodes and maintains data consistency on these copies by employing a token ring that enforces a total ordering of all data updates. The system also benefits from a membership protocol to handle network or node failure and recovery. We implement the system by enhancing a Java Virtual Machine to duplicate objects across nodes, handle consistency, and maintain fault tolerance, all transparently to the application developer.

Categories and Subject Descriptors: C.2.4 [**Computer Communication Networks**]: Distributed Systems

1. INTRODUCTION

As today's need for distributed applications becomes increasingly important, few frameworks exist to aid the developer in creating efficient, replicated objects. The ability to replicate whole objects across a heterogeneous network of machines, while always maintaining data consistency, is a common requirement shared by many distributed applications.

Communication in parallel and distributed systems is usually achieved through operations on shared or replicated objects. Common examples of such objects include shared memory, files, communication channels, and synchronization primitives like semaphores and mutexes. The semantics of operations on objects should not depend on whether or not the objects are replicated. However, if an object is replicated, then the operations on this object must satisfy certain consistency requirements.

In this paper, we use *sequential consistency* as our model of consistency. This has an advantage over the stronger model of *atomic consistency*, in that it gives rise to more efficient implementations of replicated objects. In Section 2, we demonstrate that sequential consistency can still be maintained even if partially non-blocking (and hence more efficient) implementations of replicated objects are used.

Clearly, supporting synchronization and maintaining the consistency of replicated objects presents a substantial challenge. For the application developer, this translates to a significant amount of time and energy invested into maintaining the consistency requirements of the model—instead of focusing on the application itself.

This paper proposes a framework, Java Replicated Objects (JRO), that provides synchronization and consistency among replicated objects. We choose to support the framework using the Java language; hence, our framework has the potential to replicate objects across different platforms.

While Java itself provides very loose mechanisms for exploiting parallel computing and shared memory, it does not yet provide any mechanism for replicating encapsulated data as objects. There exist Java toolkits that allow objects to be copied across computers (Java Shared Data Toolkit); these toolkits are limited in that updates to a shared object in one node are not seen by copies of the object residing in other nodes. Other toolkits, such as Java RMI, allow objects to reside on one host but be modifiable from other nodes; the shortcoming of this model is that it suffers from a single point of failure. If the host where the object resides fails, then the object is lost for the entire system. We address these shortcomings with the JRO system.

In JRO, the replication of objects is transparent to the application programmer. In fact, since

Authors' address: X. Ke, H. Pham, S. Sen and A. Solis, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139; email: {xke, hubert, sidsen, aosolis}@mit.edu

the replicated objects (and target nodes) are specified during program development, a node that has received its copy of a replicated object can further redistribute that object to other nodes. In other words, the object replication mechanism does not require each node to be aware of all other nodes in possession of the same object. This maintains both a high level of redundancy and fault tolerance in the system, as well as better transparency for the application programmer.

The use of replicated objects by itself improves the fault tolerance of a system. However, the benefits of this replication are lost if the system is unable to properly resolve inconsistencies that result from failures in the network. To this end, we require that JRO be resilient to link and other connectivity errors, as well as continue to operate when faced with single-node or multi-node failures or restarts. Furthermore, the system should be able to recover from token loss in a way that preserves consistency with respect to concurrent sets of events. To achieve this, we implement a membership protocol designed specifically for use in a token ring network. Our protocol is inspired by the Totem single-ring membership protocol [Amir et al. 1995].

This paper is organized as follows. Section 2 describes the consistency model of our system. Section 3 details the design of the messaging subsystem, and Section 4 explains the fault tolerance mechanisms of the membership protocol. In Section 5, we present an overview of our implementation of JRO, and evaluate its performance in Section 6. We summarize related work in Section 7 and conclude in Section 8.

2. CONSISTENCY

The existence of a correct and efficient memory consistency model is essential in any parallel system. One of the most intuitive memory consistency models is that of sequential consistency. Sequential consistency provides users with a single view of a distributed memory, by requiring that 1) program order is maintained by individual processors, and 2) a single sequential order exists among operations from all processors.

The choice of a memory consistency model greatly affects system efficiency and performance. We briefly describe our system model and explain our rationale for adopting a sequentially consistent memory model that transmits only write events, and blocks on certain read events.

2.1 System Overview

At the highest level, our system consists of a set of processes, a set of objects, and a set of events on these objects. The processes or nodes are arranged in a "ring" formation, with each node in the ring adjacent to two neighbors.

For simplicity, our system considers only write and read events on memory. Messages alerting a write or read event can be sent to network nodes across the ring. The vehicle for the transmission of messages is a *token*. During normal operation, only one token exists on the ring at any given time. The token is passed from one neighboring node to another, continuously traversing the ring and visiting every node on the network with each revolution. The token always traverses the ring in the same direction.

2.2 Blocking Token Algorithm

Consider first a system where nodes may only read and write to shared memory when they have the token. While a node does not have the token, it cannot compute, and must block until it receives the token again. Upon receiving a token, the node unblocks and performs all the write operations that have been previously logged on the current revolution of the token. The node then holds the token for a predetermined amount of time, during which it is free to perform read and writes to distributed shared memory.

In this scenario, a total ordering is imposed on all the read/write events in the system. Every node performs all operations logged by the other nodes in the order they were listed on the token, and logs its own operations to the token as they are executed. This guarantees that all the nodes perform operations in the same order—the order the operations were logged onto the token. This satisfies the requirements of sequential consistency.

To maintain sequential consistency, it is not necessary to transmit both read and write messages. Specifically, the nodes on the network need not be aware of read events on other nodes, since these events have no effect on state. Eliminating all read messages significantly reduces network traffic and improves performance without sacrificing correctness of implementation.

2.3 Improved Token Algorithm

So far, the implementation requires that nodes stop all processing whenever they do not have the token. This prevents the system from exploiting parallelism fully, because at most *one* node can be performing operations at any given time. We show how to relax the requirements of the system to allow nodes to execute some types of operations without blocking, even if they do not have the token.

Definition 2.1. The relation $\rightarrow$ on the set of operations of a system is the smallest relation satisfying the following two conditions: (1) If a is on the token before b is appended to the token, then $a \rightarrow b$. (2) If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Note that unlike Lamport's happens-before relation, $\rightarrow$ is a total ordering.

Definition 2.2. A set C of operations is concurrent if for any two operations $a, b \in C$ from two different processes, $\neg(a \rightarrow b)$. If C_1 and C_2 are concurrent sets (CCS), we say that C_1 precedes C_2 or $C_1 \rightarrow C_2$ if there exist $e_1 \in C_1$ and $e_2 \in C_2$ such that $e_1 \rightarrow e_2$, and there does not exist $e_3 \in C_1$ and $e_4 \in C_2$ such that $e_4 \rightarrow e_3$. A sequence of disjoint concurrent sets $C_1, C_2, \ldots, C_k$ is called a chain if they contain all the events in a computation, and if for any $x, y, 1 \leq x, y \leq k : \; C_x \rightarrow C_y \iff x < y$.

The distributed computation is modeled as a chain of concurrent sets (CCS). We generate concurrent sets through an on-line algorithm that uses a circulating token to bring operations to all processes. The algorithm is shown in Figure 1. The algorithm returns a concurrent set with every round trip of the token. The counter c_1 is used to determine when the token has completed a trip. When a concurrent set is determined, counter c_2 informs other processes that a set is available. The set S_1 accumulates operations that will form a concurrent set and S_2 is used to keep the latest concurrent set. The local variable CCS is assigned a concurrent set whenever one is returned. Q_1 queues the operations at each node; its contents are added to the token when it arrives.

Read operations that are requested after a concurrent set is processed but that are not preceded by writes on the same process are non-blocking. They immediately return a value based on the state of the object after the previous concurrent set. Write operations are also non-blocking because they do not return a value. Read operations that are requested after one or more writes since the last concurrent set are blocking. They will be processed with the next concurrent set.

As a result of these rules, each process sees a common history of execution consisting of a chain of concurrent sets interspersed with local read operations. Given a concurrent set C_k, let $S_k = \{W_{C_k}, R_k\}$ be the set of operations consisting of all the writes in concurrent set C_k and all local reads requested after C_{k-1} was processed but before C_k was received. The system then executes operations in S in an order that satisfies the $\rightarrow$ relation. In particular, this scheme guarantees that all nodes execute write operations, which affect global state, in a common order.

3. MESSAGING

In order to support data sharing across networked nodes, and to facilitate a membership protocol that ensures fault tolerance, we employ a message-passing mechanism. The messages are delivered on a token that is continually passed between all nodes in the ring.

Each node ultimately knows the addresses of all members in the ring but only forms persistent connections with its two nearest neighbors. That is, every node maintains a connection with the previous node from which the token should be received, and a connection with the next node to where the token should be sent ($nextNode$). Upon startup, however, it is only necessary for the user to specify the $nextNode$ of each node, where the $nextNode$ of the last node in the ring is set to the first

Initially $Q_i = \{\}$, $1 \leq i \leq N$
Process 1 sends $Token = (1, 0, \{\}, \{\})$ to Process 2.

Receive $Token = (c_1, c_2, S_1, S_2)$
  if $(c_1 = N + 1)$ **then**
    $CCS \leftarrow S_1$
    $S_2 \leftarrow CCS$
    $S_1 \leftarrow Q_i$
    $c_1 \leftarrow 1$
    $c_2 \leftarrow 1$
  else
    $S_1 \leftarrow S_1 \cup Q_i$
    $c_1 \leftarrow c_1 + 1$
    if $(1 \leq c_2 \leq N)$ **then**
      $CCS \leftarrow S_2$
      $c_2 \leftarrow c_2 + 1$
  $Q_i \leftarrow \{\}$
  Send $Token = (c_1, c_2, S_1, S_2)$ to next process in ring.

Request operation *op*
  if (*op* is a write) **then**
    $Q_i \leftarrow Q_i \cup op$
  else if (we performed a write after the last CCS was generated)
    delay *op* by blocking
  else
    do *op*

Fig. 1. An algorithm to generate concurrent sets using a circulating token

node. All nodes open persistent TCP connections to their $nextNodes$, thus completing a "ring" of connected nodes. Any one node can be instructed by the user to start the membership protocol, which establishes the membership of the ring and commences the token-passing mechanism.

After the membership protocol executes, all nodes store the addresses of all other nodes in the ring—this information is disseminated by the protocol. The membership protocol brings resilience to node failures and loss of tokens. The protocol's fault tolerance mechanism is described in further detail in Section 4. We now describe the underlying token data structures that are used to support the membership protocol and data sharing.

3.1 Membership Tokens

There are four special data structures used by the membership protocol: the Join token, Commit token, Foreign Join token, and Token Loss token. The Join and Commit tokens are similar to the corresponding tokens used in [Amir et al. 1995].

For each token, we give a brief explanation of how the token is used and describe its most relevant fields.

Join Token (JOIN)

A JOIN token contains a list of identifiers for processors that are potential members of a new ring that is being formed. Initially, after every node has connected to its $nextNode$, the membership protocol starts the ring formation by sending out a JOIN token from a designated node. All nodes that receive the JOIN token (including the original sender) append their node identifier to the member list in the token and pass the token to their $nextNode$.

A new ring always forms when the system is first started, but this may also occur if the membership of the current ring changes in any way; such a situation can arise both from network failure (e.g., a failing link or node) as well as network recovery (e.g., a recovered node trying to reenter the ring).

Each JOIN token has a $tokenSeq$ field that is present in all other tokens as well, with the exception of the Token Loss token. The token sequence number is an integer that is incremented every time a token is received and passed on by a node. The field is necessary for the membership protocol to determine which node has the most recent ring token, in the event of token loss.

Commit Token (COMMIT)
After the JOIN token is passed to all nodes and returns to the originating node, the token contains a full list of all operational nodes in the ring. This list becomes the final membership list for the ring, and it is disseminated to all other nodes using a COMMIT token sent by the originating node. Upon receiving the COMMIT token, a node copies the membership list and stores it locally.

Foreign Join Token (FJOIN)
The foreign join, or FJOIN, token is employed when a single node outside an existing ring wishes to join the ring. Since the foreign node does not know whether a ring is already in place, it begins by sending a JOIN token to some node n in the current ring. When n receives the JOIN token, it detects that the token is from a foreign node (it can distinguish the token from a JOIN token used to establish the initial membership of a ring). To avoid any contention or consistency problems that could result from circulating more than one token on the ring, n waits until it is in control of the Regular ring token (see Section 3.2) before sending an FJOIN token to the foreign node. The contents of the FJOIN token are identical to those of the JOIN token above, except that the member list field contains enough information for the foreign node to insert itself after n in the ring (n adds the node ID of itself as well as its $nextNode$ to the list).

Upon receiving the FJOIN token, the foreign node determines the next node it should connect to from the contents of the member list of the token (this is the $nextNode$ of n in the current ring). The foreign node then converts the token into a regular JOIN token and sends it to this node, effectively inserting itself directly in front of n in the new ring.

Token Loss Token (TLT)
A TLT is generated by a node when two failure conditions occur: 1) the token timeout of the last token sent by the node expires, indicating that the token has probably been lost, and 2) the node's connection to its $nextNode$ has failed (a "broken pipe" condition). When condition 1 occurs, a node checks for condition 2: if it has a broken pipe, the node creates a new TLT, inserts the token sequence number of the most recent token it has received, and sends the TLT to the next node. The TLT is used by the node to determine if it holds the most recent copy of the last token to have circulated the ring. If so, it reestablishes the ring membership and resends its copy of the Regular ring token.

When a node receives a TLT sent by another node, it checks if its local copy of the most recent token has a sequence number that is higher than the sequence number of the TLT; if so, it replaces the value in the TLT with this number and passes the TLT on to the next node. It is important to note that more than one TLT may be circulating the ring at any given time; for this reason, and because TLTs are used to locate the latest ring token anyway, no token sequence number is associated with a TLT.

3.2 Message Passing Token

To pass messages containing data between nodes, a **Regular Token (REGULAR)** is used. There are a variety of message types that can be encapsulated in the Regular token to support locking, writing, and searching for shared objects, as well as sending whole objects to particular nodes. When writing to (or sending) shared objects, the message type also identifies the data type to be shared. Our system supports the sharing of booleans, byte arrays, integers, longs, doubles, and whole (serialized) objects (see Section 5.5.3).

4. MEMBERSHIP AND FAULT TOLERANCE

We consider the possibility that the network may suffer from node and link failures, making it difficult or impossible for some nodes to communicate with their neighbors in the ring. The membership protocol is responsible for detecting failures in the token ring and reconstructing new rings on which the total ordering of updates can be resumed.

4.1 Network Failures

The JRO system employs a membership protocol to detect failures in the token ring and recover from them gracefully. In the event of link or node failures, the protocol attempts to circumvent the failing nodes and reconstruct a new ring on which normal operation can resume. In the event of lost or missing tokens, the protocol uses TLTs to locate the most recent copy of the last valid ring token; the node holding this copy then reestablishes the membership of the ring and restarts the circulation of the token. The membership protocol allows new nodes (including failed nodes that have become operational again) to dynamically insert themselves into the ring without violating any consistency requirements.

The membership protocol uses the FJOIN and TLT tokens to handle new entrants into the ring and lost tokens, respectively. The contents of these tokens and their use are described in section 3.1. The protocol also makes use of several user-defined constants to determine how to behave under certain failure conditions: these include round trip timeout values for each type of token, timeouts for resending a token and reconnecting to the next node, and the maximum number of resend attempts and reconnect attempts the system should make. All of these constants are stored in a separate header file and can be easily modified by the user of the system.

In this section, we present the membership protocol as a finite state machine (FSM) with a set of states and a set of events that cause transitions between the states. The FSM is shown in Figure 2.

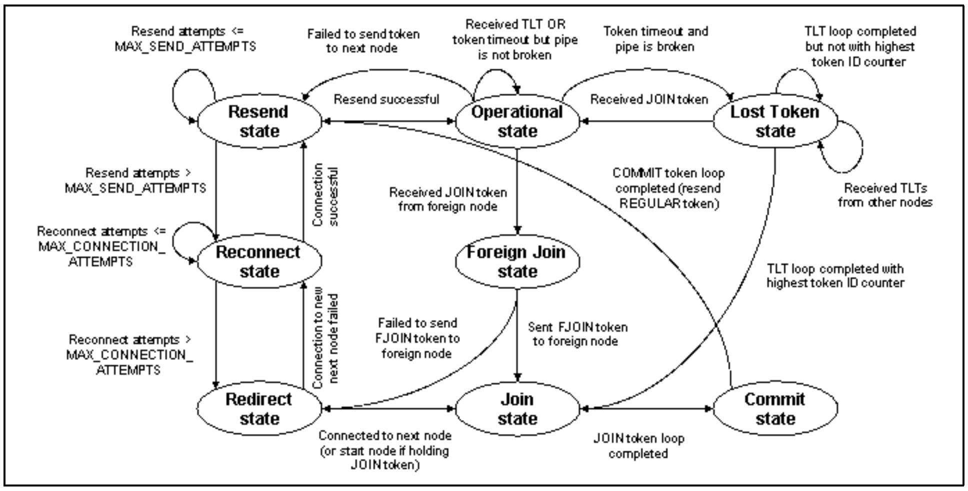

Fig. 2. The finite state machine for the fault tolerance mechanism of the membership protocol

4.1.1 *The Eight States of the Membership Protocol.* The conditions present in each of the eight states, and the events that lead to them, are shown in Figure 2. In this section, we list some of the less obvious conditions or events that apply to certain states.

(1) The Operational state encapsulates the regular, fault-free operation of the system (the REGULAR token is being passed around from node to node and updates to replicated objects are performed in total order). Failure events that occur in this state cause a node to transition to an error state and invoke the fault tolerance mechanism of the membership protocol.

(2) If a node enters the Redirect state while holding a JOIN token (which is being passed around to establish the membership of a new ring), then it automatically connects to the start node of the new ring (the originator of the JOIN token), because technically it does not know of any succeeding nodes beyond its predefined $nextNode$.
(3) When a node receives a JOIN token from a foreign node (causing it to enter the Foreign Join state), it waits until it has possession of the REGULAR ring token before sending an FJOIN token to the foreign node. This is enforced to avoid contention between multiple tokens and inconsistencies which may occur if more than one token is allowed to circulate the ring (see Section 3.1).
(4) If a node receives a TLT that it originated while in the Lost Token state, it only attempts to reestablish the ring membership (by transitioning to the Join state) if its most recent token sequence number is equal to the highest token sequence number in the TLT. Since the token sequence number monotonically increases every time a token is passed from one node to the next, this guarantees that only one node in the ring will assume the responsibility of reestablishing ring membership.

It is worth noting that since the membership protocol forces nodes to redirect their sending connections to nodes that come after it in the ring ordering, every complete CCS is eventually delivered to every operational node in the ring. By keeping track of the total number of members (in the current ring membership) at every node, the algorithm we use to determine concurrent sets (see Figure 1) guarantees that a CCS is never delivered to any node more than once.

4.2 Replicated Object Freshness

Even though the JRO system is resilient to network failure, the application programmer is still responsible for ensuring the freshness of all replicated objects that a node is interested in. In other words, it is up to the node to request up-to-date copies of these objects when it reenters the ring (e.g., after a previous failure) or when it enters the ring for the first time. Fortunately, this usually coincides with program semantics since applications will usually request the replicated objects they need at program startup.

5. IMPLEMENTATION

Each process or node in the network is a modified Kaffe Java Virtual Machine (VM) that incorporates the entire set of functionality needed to manipulate replicated objects. The modified JVM intercepts “get” and “set” operations performed on the field data of distributed objects. By doing this, we limit our problem domain to operations performed on the data fields of replicated objects; the framework does not propagate any other operations that have side-effects in a distributed object’s method.

Our installation relies upon the `Pthreads` library of non-blocking kernel thread routines. Each node maintains a unique 64 bit identifier, a 32-bit IP address, and a 16-bit port connection information to its neighbor in the token ring. The latter parameters can be specified through system parameters to the modified VM. We describe some key aspects of the implementation below.

5.1 Kaffe Runtime

We use existing Kaffe structures to maintain runtime state, including the Java heap and the stack. To keep track of replicated data, we added a Universal Unique Identifier (UUID) field to Kaffe’s representation of Java objects. The remainder of the modifications to the Kaffe source code were for intercepting the virtual machine’s byte-code interpretation of various Java instructions. The main directives that we intercept are reads and writes to memory locations through the getting and setting of field values.

5.2 ReplicationManager

In addition to the core VM interpreter, we also extended the class libraries that make up the Java runtime environment. Developers interface with the JRO system through the `ReplicationManager` class. This class has the static methods `replicate` and `unreplicate` to manage replicated objects.

The application programmer using our framework is responsible for designating objects to be replicated across specified nodes in the network. The programmer can identify which objects are to be shared among which nodes by requesting copies of the replicated objects. During runtime, changes made to an object's field data on any node are consistently reflected on all copies across the network.

Objects in JRO are not replicated unless otherwise specified. Marking an object as replicated does not create any new copies of the object, nor does it distribute it to other Java VMs—interested nodes must explicitly request copies of the object using their respective UUIDs. `Unreplicate` stops the VM from monitoring any network messages regarding the object.

5.3 Memory Management

Memory management is complicated in systems that provide remote references to objects because an object created on one machine may be referenced by a process on another machine. Network latencies make it impractical to traverse the entire object graph. However, because we keep a copy of an object on each node that participates in its replication, garbage collection remains relatively simple.

When a replicated object becomes eligible for garbage collection at a node, the node marks its copy as not replicated. We do this so that the `finalize` method does not affect the other replicas. Afterwords, finalization and garbage collection proceed normally.

5.4 Locking

A *critical section* is the part of a program that accesses shared resources. Only when a process is in its critical section can it be in a position to disrupt other processes. Programmers can avoid race conditions by making sure that no two processes enter their critical sections at the same time.

In the Java language, critical sections are identified with the `synchronized` keyword. The Java VM associates a lock with every object that has synchronized code. Before a thread enters a critical section, the Java VM sets the object's lock to prevent other threads from entering their critical sections.

In a distributed system like ours, the lock on an object must be able to prevent not only threads on the local VM but also threads on remote VMs from entering their critical sections. This is a version of the distributed mutex problem. We use the solution described in [Lamport 1978] to implement distributed locks.

5.5 Messaging Subsystem

Once the necessary data for achieving replication is determined at the VM level, this information is sent to the messaging subsystem layer to be packaged and correctly distributed among the nodes of the network. At the receiving nodes, the transmitted messages are unpacked into the VM and interpreted as if replicated instructions were executed locally.

5.5.1 *Protocol.* Our message subsystem communicates through persistent TCP connections. We chose this protocol over the broadcast network (UDP) paradigm in order to take advantage of TCP's simplicity, packet error correction and recovery, and tolerance of large packet sizes.

5.5.2 *Network Library.* We use the Networking, Messaging, Servers, and Threading Library (NMSTL) [Salz 2002] to implement the messaging subsystem. The C++ library provides sockets, basic networking, and messaging functionality. We made minor modifications to the library source code to improve its accuracy in reporting network failures.

5.5.3 *Serialization.* Objects are distributed across nodes in the network when they are initially replicated, and when they are used as data for write instructions (e.g., when a field of one object is set equal to another object). These objects must be serialized into appropriate byte arrays before they can be transmitted through the network medium. To simplify the interface between the messaging subsystem and Kaffe, object serialization is performed at the VM level.

We employed the framework provided by NMSTL to serialize Java's primitive objects. For non-primitive objects, including references and arrays, we implemented a C-based object serialization library similar to Java's `ObjectOutputStream` and `ObjectInputStream`. This library utilizes class structure and replication information within Kaffe to minimize the amount of data necessary to reconstruct the object at the receiving end of the network.

5.6 Interfacing - Threadsafe Queues

The primary means of communication between the VM and the messaging subsystem is through the *tqueue* threadsafe queue implementation of NMSTL. The VM and messaging subsystem share a common queue to store messages that should be added to the current CCS of the REGULAR ring token (the set S_1 in Figure 1). The VM pushes messages on the queue in accordance with the consistency model, and the messaging subsystem clears the queue (adding the messages to S_1 every time it receives control of the REGULAR token). The VM provides a method for the messaging subsystem to call when a CCS is ready and needs to be handled. The messaging subsystem also provides methods that the VM can call to obtain important information about the status of the ring network and current membership.

The use of threadsafe queues and simple interface methods allows the VM and messaging subsystem to communicate in an asynchronous manner.

6. PERFORMANCE

As a preliminary benchmark of the performance of our system, we wrote a simple test program that executes a loop back between the VM and the messaging subsystem. The purpose of this program is to measure the overhead of replicating objects using JRO. The program we used performs the following tests:

(1) 1 million reads followed by 1 million writes on a field of an unreplicated object

(2) 1 million reads followed by 1 million writes on a field of a replicated object.

The results of these tests are summarized in Figure 6.

	Regular Object (ms)	Replicated Object (ms)
1000000 reads	197	735
1000000 writes	247	1278

Fig. 3. Preliminary benchmarks of the JRO system, showing (roughly) a factor of 10 overhead for object replication

We are currently in the process of testing our system's performance on a novel application that calculates the individual digits of π. We use the Bailey-Borwein-Plouffe [Euresti et al. 2001] digit extraction algorithm to calculate π to an arbitrary number of digits across several nodes.

Please see `http://web.mit.edu/aosolis/www/JRO/` for updates.

7. RELATED WORK

While numerous packages exist to provide Java with a distributed shared memory, implementing a fault-tolerant distributed application still requires significant programmer effort.

The Java Remote Method Invocation (RMI) is a widely known package to create distributed Java applications. However, it does not provide true replication across processors, as it is designed to support only procedure calls on remote objects. There are packages available which extend RMI's functionality [Microsystems 1998] [Thiruvathukal et al.] and transparency for distributed applications [Euresti et al. 2001]. However, the responsibilities of fault-tolerance and memory coherence are still left to the programmer.

Consistent replication of objects is a considerably more challenging problem. Manta [Maassen et al. 2001] approaches the problem by rewriting the Java compiler and RMI run time system to optimize sharing. Manta's Replicated Method Invocation (RepMI) offers object replication on top of its native compiler. In our system, no changes are necessary to the Java virtual machine.

Our membership protocol is inspired by the Totem Single-Ring/Membership Protocol [Amir et al. 1995]. While Totem provides consistent concurrent operations over distributed nodes, it does so over a broadcast network, implementing their system with UDP. Our system simplifies the process with implementation through TCP, eliminating the need for manual sequencing of packets. Hence, our system's membership protocol is adapted for a network with persistent connections between nodes rather than a broadcast network.

8. CONCLUSION

The development of a platform to support parallel computing continues to be an active area of research. Java's portability, simple user interfaces, built-in support for threads, and object-oriented structure make the language well-suited for the task.

To this end, we have built a framework that frees the developer from synchronization and consistency issues associated with distributed applications. We employ a sequential consistency model for the operations on replicated objects to maintain correctness.

The framework itself is implemented by modifying a Java Virtual Machine and creating an underlying messaging subsystem. Our JVM handles object replication and, along with the messaging subsystem, enforces the consistency model. The messaging subsystem also handles network communication and token-based message passing. To ensure that our system is resilient to network or node failure, we design and implement a membership protocol to handle joining and failing nodes dynamically.

Our solution is currently dependent upon several packages specific to the Unix platform. We sacrifice this portability for simplicity of implementation. Whenever possible, we take advantage of existing data structures within the Kaffe virtual machine and link into general libraries such as NMSTL. By working at the level of the virtual machine, we achieve functionality without changing the semantics of the Java language or dealing with the complexities of the compiler.

To make use of our system, the application developer adds only one call to a static method of the `ReplicationManager` for each object he wishes to replicate; this represents a tremendous savings in time and effort.

With our framework, the variety of possible distributed applications is endless. A worthwhile endeavor for future work would be to explore how the JRO framework benefits highly parallel, shared-memory applications that require a certain degree of fault tolerance during execution.

REFERENCES

Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., and Ciarfella, P. 1995. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems 13,* 4, 311–342.

Euresti, D., Saraswat, A., Tellex, S., and Wu, L. 2001. Dude, where's my object? A system for sharing java objects.

Keleher, P., Dwarkadas, S., Cox, A. L., and Zwaenepoel, W. 1994. TreadMarks: Distributed shared memory on standard workstations and operating systems. 115–131.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21,* 7, 558–565.

Maassen, J., Nieuwpoort, R. V., Veldema, R., Bal, H. E., Kielmann, T., Jacobs, C. J. H., and Hofman, R. F. H. 2001. Efficient Java RMI for parallel programming. *Programming Languages and Systems 23,* 6, 747–775.

Microsystems, S. 1998. JavaSpace specification.

Salz, J. 2002. NMSTL for C++ 0.5. URL: http://nmstl.sourceforge.net/.

Thiruvathukal, G. K., Dickens, P. M., and Bhatti, S. Java on networks of workstations (JavaNOW): A parallel computing framework inspired by Linda and the Message Passing Interface (MPI).

Yu, W. and Cox, A. L. 1997. Java/DSM: A platform for heterogeneous computing. *Concurrency - Practice and Experience 9,* 11, 1213–1224.