# Higher-Order Distributed Objects

HENRY CEJTIN, SURESH JAGANNATHAN, and RICHARD KELSEY
NEC Research Institute

We describe a distributed implementation of Scheme that permits efficient transmission of higher-order objects such as closures and continuations. The integration of distributed communication facilities within a higher-order programming language engenders a number of new abstractions and paradigms for distributed computing. Among these are user-specified load-balancing and migration policies for threads, incrementally linked distributed computations, and parameterized client-server applications. To our knowledge, this is the first distributed dialect of Scheme (or a related language) that addresses lightweight communication abstractions for higher-order objects.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; *extensible languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*

General Terms: Experimentation, Languages

Additional Key Words and Phrases: Concurrency, continuations, higher-order languages, message-passing

## 1. INTRODUCTION

Process communication in a distributed environment requires solutions to a number of important, and potentially troublesome, issues. These issues concern the interaction of parallel and distributed tasks with local, sequential computation, efficient migration of processes across nodes, effective storage management for long-lived distributed applications that create data on many different nodes, and the implementation of protocols for a loosely coupled environment. The problems are old, and many solutions have been proposed [Andrews 1991; Mullender 1993]. Almost all these solutions, however, either entail defining a new programming language or adding special primitives to an existing language to handle concurrency, distribution, and communication. In general, the latter proposals have used base languages that typically provide little support for defining or composing new abstractions. As a result, the semantics of new primitives to handle distributed programming cannot be easily expressed in terms of operations already provided by the language or easily combined together to provide new functionality.

Our primary goal in this article is to define a small set of language abstractions for distributed computing that enable a variety of applications with different commu-

Authors' address: 4 Independence Way, Princeton, NJ 08540; email: {henry; suresh; kelsey}@research.nj.nec.com.

nication and computation requirements to execute efficiently using stock hardware and interconnect technologies. Keeping the distributed programming interface orthogonal to the sequential programming interface is an equally important design goal. To realize this goal, we must choose a sequential base language in which these abstractions can be easily expressed.

High-level languages such as Scheme [Clinger and Rees 1991] and ML [Milner et al. 1990] provide support for data abstraction via first-class procedures and support for sophisticated control abstraction via first-class continuations [Haynes and Friedman 1987]. Because of their generality, procedures and continuations can be used to express operations for parallel and distributed programming. Using them as building blocks for high-level concurrent and distributed abstractions avoids the need to define an entirely new language or to introduce ad hoc primitives to an existing one. In this article, we develop a small set of such abstractions using Scheme as the base language.

The distributed system described here (called Kali Scheme[1]) supports concurrency and communication using first-class procedures and continuations. Since efficient load balancing is an especially important problem on systems in which multiple users with different workload requirements share resources with long-lived distributed applications, our implementation transmits procedures and continuations efficiently.

We integrate procedures and continuations into a message-based distributed framework that allows any Scheme object to be sent and received in a message. This integration engenders a number of new abstractions, paradigms, and implementation techniques for distributed computing; some of these are enumerated below and discussed during the remainder of the article:

*User-Level Load Balancing and Migration.* Given the ability to transmit continuation objects, and given a representation of threads in terms of continuations, thread schedulers residing in different address spaces (presumably residing on different machines in a network) can migrate tasks from one address space to another; these schedulers may be implemented entirely as user-level procedures in Scheme. Migration policies can be implemented without modification to application programs.

For example, consider a parallel program $P$ running on a network of workstations shared among a set of users. Applications created by the owner of a workstation are assumed to have priority over tasks created by $P$. Consequently, whenever a user initiates an application on node $N$, some set of $P$'s tasks currently running on $N$ may need to migrate to other machines. In this environment, it is important that the specific migration policy used to determine which tasks migrate be determined by $P$ and not be hardwired as part of the internal thread implementation.

Another example where user-level thread migration may be useful is in parallel query searches over a distributed database. A query may spawn a collection of tasks, each of which are responsible for scanning some portion of the database. If the entries in the database are related under some well-understood ordering, it may

[1]Kali is a fearsome goddess of the Hindu pantheon with one thousand arms, each equipped with a distinct weapon; we chose the name because of its relevance to distributed programming.

be possible for a task to determine that the input query will not be satisfied in the piece of the database it is currently examining. In this case, the task should ideally migrate to a more-interesting section of the database. The choice of where to go next is determined by a scheduler and is highly dependent on the structure of the database itself.

*Incremental Distributed Linking.* The system performs dynamic linking of code objects sent in messages. Distributed programs can thus be incrementally compiled and linked on a per-address-space basis; linking a new program does not entail a networkwide broadcast.

An important example where distributed linking would be useful is in distributed debugging and monitoring. Consider a long-lived distributed application running on a collection of address spaces. To monitor facets of the computation dynamically, Kali Scheme users can input a monitoring procedure directly to a read-eval-print loop or load a file containing the procedure and apply it on different nodes *without* halting the current application. In other words, users can define probes to survey the state of an ongoing computation; these probes do not need to be defined at the time the computation is instantiated.

*Parameterized Client-Server Applications.* The ability to communicate higher-order objects in messages permits servers and client applications to establish new protocols and functionality dynamically; this capability is especially important for wide-area distributed applications that would benefit from clients being able to off-load functionality to servers dynamically.

In Section 5.2, we sketch a simple example where such functionality is useful. The example assumes a server controlling a display and a client that manages a large database. Queries on the database result in modifications to the display. To optimize communication between client and server, and to minimize latency in refreshing the display, we require the ability to ship portions of the database dynamically along with relevant accessor procedures to the server.

Cooperative programs or multiuser dungeon (MUD) environments also require this kind of functionality [Curtis 1992]. A MUD is a network-accessible program and database which can be extended by many users in real time. The essential characteristics of a MUD are that it be extensible from within; in other words, users can add new data elements to a MUD database, or new functionality to the MUD core program, and have these extensions made available to other users working in the same environment.

*Long-Lived Parallel Computation.* Process state can be freely and transparently migrated, and processors (mostly asynchronously) can perform local garbage collection. These features make the system well suited to express long-lived parallel applications. The ability to migrate process state freely gives applications the freedom to build customizable schedulers; the benefits of such a capability were described above.

Automatic garbage collection is essential for distributed client-server applications such as the one given in Section 5.2 where new data is constantly being created and discarded on remote nodes. Garbage collection is also vital for applications such as the parallel search program described earlier. Tasks created by this application may create temporary data structures to hold intermediate results of the search; these results may refer to data elements recorded on other nodes. The memory allocated to these intermediate structures need not be explicitly recycled by the application. The benefits of automatic garbage collection for this application are clear: data structures can be freely created without concern for space leaks or memory limitations.

*Distributed Data Mining.* A data-mining program is one that examines elements of a database and attempts to infer interesting relationships about these elements. A distributed data-mining application is one which operates over a distributed database.

Complex data-mining applications that monitor and examine distributed databases need to ship procedures to different nodes as new inferences are generated. A task within an application that encounters an interesting feature in some portion of a database may need to notify other tasks or create new tasks on other nodes in response. Lightweight communication protocols that can transmit and receive programs are useful in realizing this kind of functionality.

Consider the following problem: given a large and complicated mass of input data, one requires a system that performs two separate but related tasks: (1) It will act as a heuristic database, ready to accept user queries either about the status of a particular element or the likelihood of some more-complex phenomenon given the current state. (2) It will act as a monitor and alarm system, posting notices when significant state-changes occur. In determining the likelihood of more-complex states, the system might use quantitative tests, heuristic decision procedures (as rule-based systems do, for example), or any mix of the two. If the input data is recorded on different address spaces, monitor and alarm tasks will themselves need to be distributed. Because the database is assumed to be constantly evolving, resolving user queries will require creating tasks on those address spaces where the required data is recorded, rather than having the data sent to the node containing the task making the query.

*Executable Content in Messages.* There continues to be growing interest in safe electronic delivery of component software and in allowing executable content to be sent in messages on wide-area networks. The World-Wide Web, for example, is a large distributed system built on top of a wide-area network. Its expressive power would be significantly enhanced if Web browsers had the capability of easily receiving and executing programs installed at other sites.

Consider implementing a remote registration program over the Web. To register a new user of a product, we envision a program downloaded to the user's machine that collects relevant data and, only upon completion, sends the data back to the host site. As with the client-server example cited earlier, computation is performed locally, reducing message traffic and improving latency.

As another example, consider a large application program available on some remote host. In a conventional distributed system, running this program locally requires downloading its entire image. This is potentially wasteful if only a small piece of its overall functionality is required; for example, libraries linked to the application should only be downloaded when actually used. An alternative solution is to have the application dynamically fault procedures on demand as the application runs.

The system we describe consists of a collection of address spaces within which potentially many concurrent, lightweight, preemptible threads may execute. We envision address spaces as uniformly distributed among different nodes in a network ensemble of (potentially) heterogeneous machines. The execution environment for the prototype described in this article is a network of workstations running Unix. Although the implementation makes no assumption on whether the underlying network is tightly or loosely coupled, we expect Kali Scheme to be best utilized in a loosely coupled wide-area network.

Address spaces are first class and may be generated dynamically. Threads within an address space may communicate with one another via shared memory; communication between address spaces takes place using explicit message passing.

In the next section, we present an overview of the system. Section 3 gives a brief outline of Kali Scheme's concurrency and distribution primitives; these primitives are built on top of Scheme 48, an implementation of an extended dialect of Scheme. Section 4 describes the transport mechanism, focusing on how procedures, continuations, and data are sent in messages. Section 5 presents several examples drawn from the list enumerated above; all of these examples have been implemented and tested on heterogeneous platforms including single-processor workstations, networks of PCs, and tightly coupled shared-memory multiprocessors. (The shared-memory implementation maps each address space to a logically different processor.) Section 6 outlines the garbage collector. Section 7 gives base performance figures, and Section 8 describes related work.

## 2. OVERVIEW

Many distributed systems allow complex data structures to be sent between nodes or address spaces, but few provide the same flexibility for procedures, dynamically created code objects, or other complex pointer-based structures. In the context of a language such as Scheme, efficient implementation of such functionality is essential. The system we describe in this article provides precisely this expressivity.

As we describe in the next section, threads are represented in terms of continuations. Consequently, they can also be sent in a message; transmitting a thread in a message from address space $A$ to $B$ effectively migrates the thread from $A$ to $B$. Ordinarily, continuations (and thus threads) may be closed over a large amount of state. To avoid the overhead that would occur if continuations were transmitted naively, we support a generalization of *computation migration* [Hsieh et al. 1993] to fault continuation frames lazily between address spaces; this feature significantly reduces the overall communication bandwidth requirements needed to support migration.

Since we are interested in supporting dynamic distributed applications, our implementation also allows templates to be sent in messages as well. (A template corresponds roughly to the code segment of a procedure. It contains a vector of Scheme 48 byte-codes, along with a vector of locations of top-level definitions. A template along with an environment defines a closure.) The ability to send templates enables distributed applications to be linked incrementally. An incremental copy mechanism is employed for templates, thus reducing the cost of migrating code objects.

Because we allow complex data structures to be sent in messages, and we impose no restrictions on where objects can be sent and copied, the implementation also includes a novel, distributed, (mostly asynchronous) garbage collector (GC) to collect objects referenced remotely; the collector also works over distributed cyclic structures. Local collections can run asynchronously and independently of GC's occurring in any other processor. Synchronization is only required to collect cyclic structures that span address spaces. Unlike other distributed garbage collection schemes (e.g., Birrell et al. [1993] and Lang et al. [1992]), exporting local or forwarded references to other address spaces involves minimal GC-related bookkeeping; sending a remote reference requires simply setting a bit locally in the sender. The collection algorithm also introduces no overhead for forwarded references.

Fault tolerance and security are are two important design issues we do not address in this article. Because the communication primitives we define make no guarantees about communication latency and because synchronous message sends do not implement time-outs or timestamps, a message sent to a failed node will appear to the sender as simply a high-latency operation. Moreover, the implementation does not replicate or log data on multiple nodes or stable storage. Our implementation assumes a reliable network and does not preserve timestamps or other logging information to validate message receipts.

Because Scheme is a type-safe language (i.e., operations can never be applied to objects of inappropriate type, and casts between objects of unrelated type can never be performed), the system already provides a measure of security that would not be available in distributed extensions of type-unsafe languages such as C. In addition, because Scheme is lexically scoped, closure objects sent across nodes cannot corrupt any objects on the receiver other than those that are explicitly shared. However, our system does assume trusted channels. Thus, receivers do not make any attempt to validate the integrity of incoming messages, and senders do not encrypt outgoing messages. It is of course possible to incorporate encryption and validation facilities on message-sends and receives to provide an extra level of security, but such extensions orthogonal to the main design goals of the system.

## 3. LANGUAGE ABSTRACTIONS

### 3.1 Scheme 48

Kali Scheme is implemented as an extension to Scheme 48 [Kelsey and Rees 1994], an implementation of Scheme [Clinger and Rees 1991]. Scheme is a lexically scoped dialect of Lisp. Scheme 48 is based on as byte-coded interpreter written in a highly optimized, restricted dialect of Scheme called Pre-Scheme, which compiles to C. Because of the way it is implemented, the system is very portable and is reasonably efficient for an interpreted system.[2] Unlike other Scheme implementations,

[2]Scheme 48 is roughly 10-15 times slower slower than a highly optimized Scheme compiler generating native code [Kranz et al. 1986].

```
(define-record-type thread
  ...
  continuation
  ...)

(define current-thread ...)

(define (spawn thunk)
  (let ((thread (make-thread)))
    (set-thread-continuation! thread
                              (lambda (ignore)
                                (thunk)
                                (terminate-current-thread)))
    (context-switch thread)))

(define (context-switch thread)
  (add-to-queue! runnable-threads current-thread)
  (switch-to-thread thread))

(define (switch-to-thread thread)
  (call/cc (lambda (k)
             (set-thread-continuation! current-thread k)
             (schedule-thread thread))))

(define (terminate-current-thread)
  (schedule-thread (another-thread)))

(define (schedule-thread thread)
  (set! current-thread thread)
  (let ((cont (thread-continuation current-thread)))
    (set-thread-continuation! current-thread #f)
    (cont #f)))
```

Fig. 1. A simplified implementation of threads in terms of operations on continuations.

Scheme 48 has a well-developed module system that forms an integral part of the language and system environment.

Scheme 48 supports concurrency using lightweight, preemptible threads; threads synchronize using locks and condition variables. Scheduling, blocking, and resumption of threads is defined in terms of operations on continuations [Haynes and Friedman 1987]. In Scheme, a continuation is reified as a procedure that, when applied, performs the remaining computation. To resume a blocked thread involves invoking the continuation representing the thread. Context switching is similarly implemented in terms of capture and invocation of continuations. Figure 1 sketches the definition of various thread operations, omitting certain important details regarding interrupts, synchronization, and debugging.

In the implementation shown, a thread is a simple record that holds a continuation. When a new thread is spawned, the continuation of the currently running thread is enqueued, and the thunk associated with the thread is evaluated (a thunk is a procedure of no arguments). If the currently running thread finishes its timeslice, relinquishes control, or blocks on I/O, its continuation is saved, and a new thread is scheduled; this scheduling operation restores the old continuation, reinitializes the continuation slot of the thread to be run, and applies the continuation. The reinitialization operation ensures that the garbage collector will not trace old continuations. Extending this formulation to handle timer interrupts complicates the design slightly. The system uses nonblocking I/O system calls to ensure that a thread blocked on I/O does not cause the thread scheduler itself to block.

The cost of context-switching threads is mitigated because the system uses a *stack cache* [Kelsey 1992] to make restoring and capturing continuations cheap. A stack cache stores only the most-active portion of the stack; older frames are restored on demand from the heap.

### 3.2 Extensions to Support Distribution

Scheme 48's thread system provides an expressive concurrency model, but all threads execute logically within a single address space. Consequently, while the thread implementation can be easily extended to support parallel applications in a shared-memory environment, the thread system does not contain any message-passing or communication abstractions necessary for a distributed-memory system. *Address spaces* and *proxies* are two extensions to Scheme 48 used to support distribution. We describe these abstractions in detail below.

A distributed Scheme 48 program executes within a collection of address spaces, each of which can run on a different machine in a network ensemble or multiprocessor. More than one address space can reside on a node at a time. Thus, although there will usually only be as many address spaces as there are physical nodes, programmers can create more if they wish. Threads executing within an address space can communicate using shared memory, locks, and other abstractions provided by the thread system. Communication between address spaces takes place via message-passing abstractions described below.

The manifest separation of intraaddress space and interaddress space communication is intentional. An important goal of our design was to provide programmers great flexibility to control and specify communication costs. Systems which implement logical shared memory [Cox et al. 1994; Li and Hudak 1989] on physically distributed platforms hide such costs; there is no visible source-level distinction between a local data access and a remote one. In the absence of explicit hardware support, or modifications to the underlying operating system, such approaches tend to be costly and inefficient for many kinds of programs. There are several important examples where the ability to have tasks communicate data explicitly even when executing within a shared-memory environment is useful [Kranz et al. 1993].

On stock hardware and interconnects, systems that make communication explicit are likely to exhibit better performance than those that provide only a software-based, shared-memory abstraction. Historically, however, such message-passing systems have been difficult to write and debug because many complicated issues (e.g., data placement, locality, communication patterns, etc.) must be programmer specified. In addition, the integration of distribution primitives into a language often entails modifications to the semantics of the sequential core or imposes limitations on the kind of objects that can be communicated.

The extensions described here alleviate much of the complexity found in other distributed programming languages and systems. These extensions are completely orthogonal to other Scheme primitives and can be combined and abstracted in many different ways. There are no restrictions on the objects which can be sent in a message or on the contexts in which message-passing operations may occur. As a result, many kinds of useful communication abstractions and paradigms can be defined easily. In the remainder of the article, we provide simplified code fragments of various procedures found in the system to help describe details of the implementation.

3.2.1 *Address Spaces.* Address spaces are implemented as ordinary Scheme data structures. To initialize the address spaces in a system, our current implementation associates with each address space a unique identifier and a list of pairs of *channels*; each element in the pair represents an input and output connection to another address space. Thus, every address space has a bidirectional connection to every other. Our implementation assumes the existence of operating system services for network connectivity via sockets, ports, or similar abstractions.

There are two primitive operations that return address spaces as their result:

(1) `(current-aspace)` returns the address space on which this operation is executed.

(2) `(make-aspace` *addr*`)` creates a new Scheme 48 process and address space on a machine with Internet address *addr* and notifies all other address spaces of the existence of this new address space. The address space created is returned as the result. Thus, address spaces can be dynamically created and linked to other address spaces in the system.

Because all address spaces have a direct connection to every other, the number of address spaces which can be generated is bounded by the number of input and output channels provided by the underlying operating system. For our prototype, this particular topology does not pose serious constraints in validating the utility of the design. Nonetheless, it does the limit the scalability of the implementation.

It is straightforward, however, to relax these constraints to allow arbitrary connected graphs to be built. For example, it is possible to permit an address space to be connected to only a subset of all other address spaces and to create and destroy connections between address spaces dynamically. Such functionality complicates the garbage collection algorithm slightly and requires routing decisions to be made either by the transport layer described in the next section or by the underlying network, but in either case it does not compromise the semantics of any of the language primitives we introduce.

3.2.2 *Proxies.* In addition to the usual repertoire of Scheme objects (e.g., vectors, procedures, continuations, etc.), one can also send a *proxy* as part of a message. A proxy is a distinguished record type with two slots; the first contains a system-wide uid, and the second holds a value. Part of a proxy's uid is the address space on which the proxy was created. Because of the way it is represented, creating a new proxy does not involve global synchronization. Only the uid field of a proxy is transmitted when a proxy is included as part of a message. The value slot of a proxy need not be consistent across different spaces. In other words, a proxy defines an address-space-relative abstraction.

Given a proxy $P$, the expression `(proxy-creator` $P$`)` returns the address space in which this proxy was created; `(proxy-value` $P$`)` returns the value of $P$ in the address space in which this expression is evaluated; and `(set-proxy-value!` $P$ $v$`)` sets the value of $P$ to $v$ in the address space in which this expression is evaluated.

One can think of a proxy as a vector indexed by address space uids. Each element in this conceptual vector corresponds to the proxy's value in that address space. The implementation of proxies refines this picture by distributing the vector; there is no interaddress space communication cost incurred to access a proxy's local value. This distinction is an important one because it provides programmers the ability to distinguish local and remote data. Thus, a data structure that can be safely replicated can be encapsulated within a proxy; accesses and mutations to elements of this structure do not require synchronization or communication.

### 3.3 Sending Objects

Our system uses explicit message passing. Programmers thus have significant control over communication costs. Scheme objects can be sent in a message by copy or by uid:

(1) *Copy*: Ordinary Scheme 48 structures such as lists, vectors, closures, etc. are copied between address spaces. Within a message, sharing is fully preserved. Sharing is not preserved between messages. For example, consider a complex structure $S$. If $S$ is sent in a message to an address space $A$, a new, structurally identical copy of $S$ is constructed in $A$; Scheme "eq-ness" within this copy is honored. If $S$ is re-sent to $A$, another copy is generated; "eq-ness" is not preserved between the two copies.

(2) *Uid*: Certain Scheme 48 objects are associated with a unique identifier. Procedure templates, proxies, and symbols are salient examples. These objects are always transmitted between address spaces via uid. The receiving address space, however, may not always have a copy of the object referenced; for example, sending a symbol that is created by `(read)` will cause a uid to be sent that is unknown by the receiver. In these cases, the receiver explicitly requests the unknown object to be sent. Subsequent transmission of the same uid will not require the object to be re-sent.

We describe the implementation of messages and the mechanisms used to transmit objects in Section 4.2.

3.3.1 *Communication Primitives.* Communication between address spaces is done using the `remote-run!` procedure. The expression

`(remote-run!` $AS$ *procedure* `.` *args*`)`

spawns a new thread on address space $AS$ which applies *procedure* to *args*; the operation is asynchronous and returns immediately. The definition of `remote-run!` is given in Figure 2.

If the target address space of a `remote-run!` operation is the current address space, a new thread is simply spawned to execute the application. Otherwise, a message is constructed and sent across the channel connecting the source and target address spaces. The `send-message` procedure is responsible for writing a message to the target address space. Besides the actual message, `send-message` requires

```
(define (remote-run! aspace proc . args)
  (if (eq? aspace (current-aspace))
      (spawn (lambda () (apply proc args)))
      (send-message 'run
                    (cons proc args)
                    (aspace-channel aspace)
                    (aspace-lock aspace))))

(define (send-message type message channel lock)
  (let ((message (encode type message)))
    (with-lock lock
      (channel-write message channel))))
```

Fig. 2. Procedures can be instantiated across address spaces.

```
(define (remote-apply aspace proc . args)
  (if (eq? (current-aspace) aspace)
      (apply proc args)
      (let* ((condvar (make-condvar))
             (proxy (make-proxy condvar)))
        (remote-run! aspace
          (lambda ()
            (remote-run! (proxy-creator proxy)
              (lambda (v)
                (set-condvar! (proxy-value proxy) v))
            (apply proc args))))
        (condvar-ref condvar))))
```

Fig. 3. Synchronous send/reply communication using `remote-run!`.

knowing the type of message being sent; in this case, the type is "`run`." It also needs to know the appropriate output channel and requires access to a lock to prevent other messages from being sent to the target by threads concurrently executing on its address space while the transfer is underway. `Send-message` linearizes the message and associated data and releases the lock when complete. We describe the decode operation executed on the receiving side in Section 4.2.

We can express an RPC-style version of `remote-run!` as shown in Figure 3.[3] The `remote-apply` procedure applies its `proc` argument to `args` on the target address space. The thunk sent to the target is closed over a proxy; thus, evaluating the thunk on the target requires sending the proxy as well. The value of this proxy on the source address space is a condition variable. The sending thread blocks on `condvar` until the receiver executes a `remote-run!` operation back on the sender's address space. This operation is executed only upon the completion of the application of `proc` to `args`. The target determines the source address space by evaluating `(proxy-creator proxy)`. Because the procedure executed by the target uses a con-

[3]This implementation uses condition variables. Condition variables in Kali Scheme are synchronizing cells similar to I-structures [Arvind et al. 1989]. An attempt to read a condition variable that does not have a value causes the accessing thread to block. Threads blocked on a condition variable are awakened when the variable is set.

```
(define make-handle make-proxy)

(define handle-creator proxy-creator)

(define (handle-value h)
  (remote-apply (handle-creator h) proxy-value h))

(define (set-handle! h v)
  (remote-apply (handle-creator h) set-proxy-value! h v))
```

Fig. 4. Proxies can implement network handles.

```
(define (handle-test-and-set! h pred? v)
  (remote-apply (handle-creator h)
    (lambda ()
      (if (pred? (proxy-value h))
          (set-proxy-value! h v)))))
```

Fig. 5. First-class procedures allow extensions to be easily added to the basic set of communication abstractions.

dition variable encapsulated within a proxy to guarantee proper synchronization, this implementation is concise and efficient. Note that `remote-apply` simply applies `proc` to `args` when its `aspace` argument is the same as the current address space. Thus, `remote-apply` incurs no message communication overhead in the case where the target address space is the same as the source.

Since proxies have a unique owner, they can be easily used to implement shared global data structures such as network handles [Birrell et al. 1994] as shown in Figure 4. A network handle is effectively a proxy. To extract the value of a network handle, we simply execute a `remote-apply` operation and apply the `proxy-value` procedure on the address space on which the handle was created. Because of the way `remote-apply` is defined, dereferencing a handle on the same address space on which the handle was created incurs no communication overhead. Setting the value of a handle is defined similarly.

The use of first-class procedures simplifies the specification of handles significantly. In their absence, a remote reference would require predetermined handlers to accept different requests. Generalizing the functionality of the network handle example would thus be problematic. For example, consider the definition of a `handle-test-and-set!` procedure that sets the value of a remote handle only if its current value satisfies a supplied predicate. Such a definition can be defined easily in terms of `remote-apply` as shown in Figure 5. The `pred?` argument is a procedure that is applied to the handle's value on the handle's creator address space. Figure 6 illustrates the semantics of these abstractions.

3.3.2 *Encapsulating Communication.* Handles provide a convenient shared-data abstraction. However, because the value a handle encapsulates can be retrieved only by evaluating a `handle-value` operation, using handles may sometimes cause

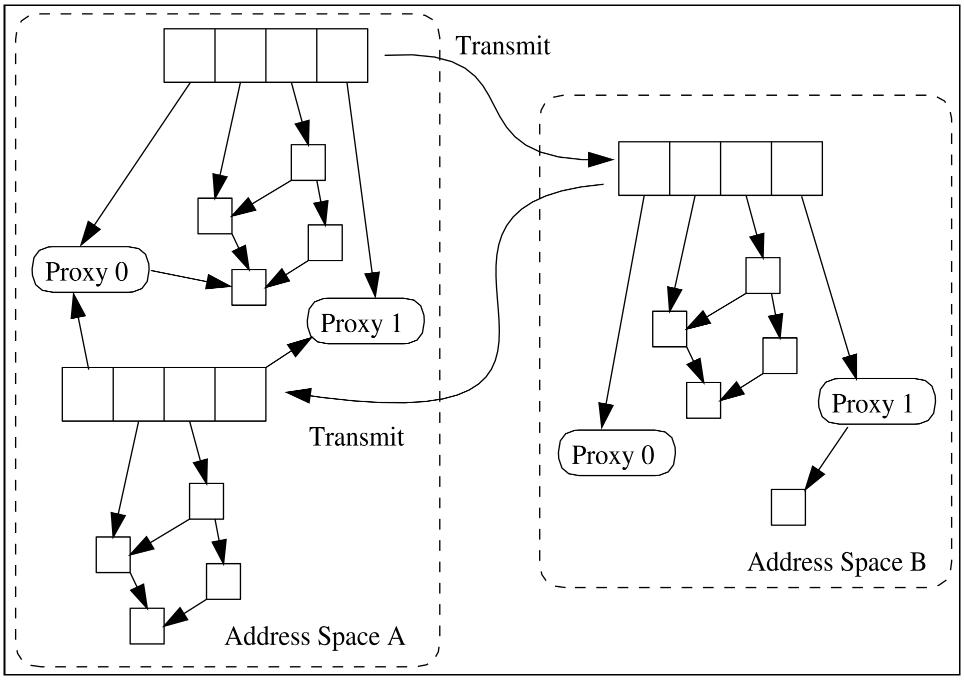


Fig. 6. Communication across address spaces. Address space A contains a Scheme structure with references to other Scheme structures and two proxies. When this structure is transmitted to another address space B, copies are made of all its fields, with structure sharing fully preserved. Of the two proxies copied, one has no local value, while the other refers to another object local to address space B. When this same structure is re-sent back to address space A, the structure is again copied. Proxies retain their identity on transmission, but new copies are made of other Scheme objects.

```
(let ((count 0)
      (lock (make-lock))
      (counter (lambda () (with-lock lock (inc! count)))))
  ...
  (let ((f  (lambda () ...  (counter) ...)))
    (remote-run AS f))
  ...)
```

Fig. 7. Procedures sent in messages may be closed over free variables which must be shared.

data and procedural abstractions to be compromised. For example, consider the code fragment shown in Figure 7.

In this program, the application of `counter` in `f` takes place on address space `AS`. Because of Kali Scheme's default copy semantics, free variables in `counter` (i.e., `count` and `lock`) are copied to `AS`. Thus, as written above, `counter` does not serve its intended purpose as a global increment procedure.

One can remedy the problem using handles as shown in Figure 8. In this rewritten version, `f` applies the value of the handle associated with `counter` on the address

```
(let ((count 0)
      (lock (make-lock))
      (counter (make-handle (lambda () (with-lock lock (inc! count))))))
  ...
  (let ((f (lambda ()
             ...
             (remote-apply (proxy-creator counter)
               (lambda () ((handle-value counter))))
             ...)))
    (remote-run! AS f)
 ...)
```

Fig. 8. Handles can be used to safely encapsulate free variables in procedures, but their use may require exposing details of how a procedure is used to expressions that would ordinarily not need to know.

```
(define (encap proc)
  (let ((proxy (make-proxy proc)))
    (lambda args
      (remote-apply (proxy-creator proxy)
        (lambda ()
          (apply (proxy-value proxy) args))))))
```

Fig. 9. `Encap` defines an encapsulation abstraction for procedures.

space in which `counter` was defined. All calls to `counter` now execute on the same address space, thus allowing the desired behavior to be realized.

Unfortunately, this solution requires altering `counter`'s definition. In this program, the alteration is easily accomplished because `counter`'s definition is readily apparent. It not likely, however, that this will generally be the case. Note that encapsulating `counter` within a handle is necessary not because it is explicitly provided as an argument to `remote-run!`, but because it occurs as a free variable in a procedure which is. Requiring alteration to `counter` constitutes a violation of procedural abstraction since the correctness of its definition is no longer dependent only on the contexts in which it is applied but also on the contexts in which procedures which call it are used.

We can use higher-order procedures to provide other encapsulation mechanisms besides handles that eliminate the problems noted above. For example, Figure 9 defines a procedure named `encap`. Given a procedure $P$, `encap` returns another procedure $G$ as its result. Assume the call to `encap` occurs on address space $AS$. $G$ is thus closed over a proxy created on $AS$ whose value is $P$. If $G$ is applied to arguments `args` on some other address space, a `remote-apply` operation is executed that applies $P$ to `args` on $AS$ and returns the result. Users of $G$ need not know whether it is a local procedure or a remote one.

Using `encap`, the original program fragment can be now rewritten as shown in Figure 10. The definition of `counter` remains unaltered. Regardless of the address space in which `f` is applied, calls it makes to `counter` within its body will execute on the address space in which `counter` is defined. Integrating higher-order

```
(let ((count 0)
      (lock (make-lock))
      (counter (lambda () (with-lock lock (inc! count)))))
  ...
  (let ((f (let ((counter (encap counter)))
             (lambda ()
                ... (counter) ...)))
    (remote-run! AS f)
 ...)
```

Fig. 10. Using `encap`, procedures can be effectively treated as handles without requiring their definition to be altered.

programming techniques with expressive abstractions for distributed programming allows the specification of a wide range of sharing abstractions.

## 4. THE TRANSPORT LAYER

Typically, all Kali Scheme address spaces will start with the same Scheme 48 image; this image contains definitions of predefined base-level procedures and data structures. One of these processes may also be associated with a read-eval-print loop (REPL). Users can thus interactively input new expressions, load programs, etc.Values yielded by expressions input to the REPL are *not* automatically broadcast to all other address spaces but instead are transmitted only when they are included in messages.

Message encoding and decoding are performed by the Scheme 48 virtual machine. Encoding and decoding are done by linearizing data structures using fairly standard techniques.

### 4.1 Sending Messages

Data structures in a message are encoded by building a vector that corresponds to a flat (linear) representation of the structure. Cycles are detected by explicitly marking objects when they are first visited and unmarking them after the current message has been fully scanned.

Templates and symbols are associated with unique identifiers. In general, these uids are globally known across all Scheme 48 images. Consequently, it will usually be the case that sending the uid of these objects is sufficient for them to have a well-defined meaning on the receiver.

Exceptions to this rule occur when procedures or symbols are generated dynamically. This can happen when users input new definitions to the REPL or load files. We discuss how such cases are handled below.

### 4.2 Receiving Messages

There is a set of coroutine pairs in each address space. Each element in the set is responsible for receiving and dispatching messages sent from a particular address space. The *decoder* routine receives a flattened representation of a message, interprets it to reconstruct a valid Scheme object, and sends the resulting Scheme object to a *dispatcher* that then executes the appropriate operation.

The decoder is implemented as part of the Scheme 48 virtual machine. After the object is decoded, it is dispatched based on its type. There are four basic types of messages:

(1) *User-Space Messages.* These messages correspond to `remote-run!` operations.
(2) *Request Messages.* When a message is received containing uids of objects *other than proxies* that are not mapped to any Scheme 48 value in the receiving address space, a request message is sent asking for the value.
(3) *Reply Messages.* Any user-space message that contains unresolved uids is placed on a pending queue. Reply messages communicate the values of unknown uids. Whenever a pending message becomes fully resolved, i.e., when the values associated with all uids in the message become known, the message becomes executable, and a new thread is spawned to evaluate it.
(4) *GC Messages.* Periodically, messages are sent among address spaces to garbage collect objects with global identity; proxies are a noteworthy example. We defer discussion of the GC algorithm to Section 6.

The basic structure of the dispatcher is shown in Figure 11. The heart of the dispatcher is the `process-message` procedure. It takes a message and an output channel and interprets the message:

(1) **Run** messages are used to implement `remote-run!` operations. The receiver applies the procedure to the provided arguments in a separate thread of control.
(2) **Uid-request** messages are requests for the value of an object referenced by a uid. When a message containing a uid is sent, no attempt is made by the sender to verify that the receiver does actually have the object referenced by the uid. When the decoder on the receiver encounters a message with uids that have no value in its address space, it notifies the dispatcher to initiate a uid-request message. However, *no* request message is sent for proxies.
(3) **Uid-reply** messages are received in response to a uid-request message. Upon receipt of a uid-reply, the receiver reconstructs the object on its address space, updating relevant uid tables. In addition, the count of outstanding uid requests is also decremented. When this count becomes zero, all outstanding requests have been serviced, and messages which were previously placed on a pending queue (because they referenced uids not known on this address space) can now be executed.
(4) **Pending** messages are initiated only by the decoder and are never sent between address spaces. A pending message is generated whenever the decoder encounters a message that references uids whose values are not known. The dispatcher to which the message is sent places the message on a queue and sends a uid-request message to the appropriate address space. The message is not executed until all its missing components are received.

    It is often the case that administrative messages (e.g., uid-reply) will themselves contain incomplete data. These messages will be flagged by the decoder as pending but must still be handled by the dispatcher to initiate a uid-request. For example, a uid-reply message may contain unknown location or template uids; this message will be enqueued on a pending queue by the dispatcher, removed and processed only when the values for these unknown uids are received.

Figure 12 illustrates the relationship between the decoder and dispatcher.

```
(define (dispatcher in-channel out-channel out-channel-lock)
  (let loop ()
    (let* ((size (get-message-size in-channel))
           (buffer (make-byte-vector size)))
      (read-message buffer in-channel)
      (disable-interrupts!)
      (let ((message (decode-message buffer)))
        (enable-interrupts!)
        (process-message message out-channel out-channel-lock)
        (loop)))))

(define (process-message message out-channel out-channel-lock)
  (case (message-type message)
    ((run)
     (spawn (lambda () (apply (message-proc message)
                              (message-args message)))))
    ((uid-request)
     (send-message message-type-uid-reply
                   (map make-uid-reply (uid-list message))
                   out-channel
                   out-channel-lock))
    ((uid-reply)
     (for-each process-uid-reply (uid-reply-list message))
     (with-lock pending-message-lock
       (set! *outstanding-request-count*
             (- *outstanding-request-count* 1))
       (if (= 0 *outstanding-request-count*)
           (release-pending-messages!))))))
    ((pending)
     (if (handle-pending-message message out-channel out-channel-lock)
         (process-message message out-channel out-channel-lock)))
    ((gc-request)
     (let ((proxies (get-local-proxies (message-aspace message))))
       (send-message message-type-gc-reply
                     proxies
                     out-channel
                     out-channel-lock)
       (reset-proxies! proxies)))
    ((gc-reply)
     (collect-local-proxies (message-proxy-list message)
                            (message-aspace message)))))
```

Fig. 11. Incoming messages are handled by dispatching on one of six basic types: `run`, `uid-request`, `uid-reply`, `pending`, `gc-request`, or `gc-reply`.

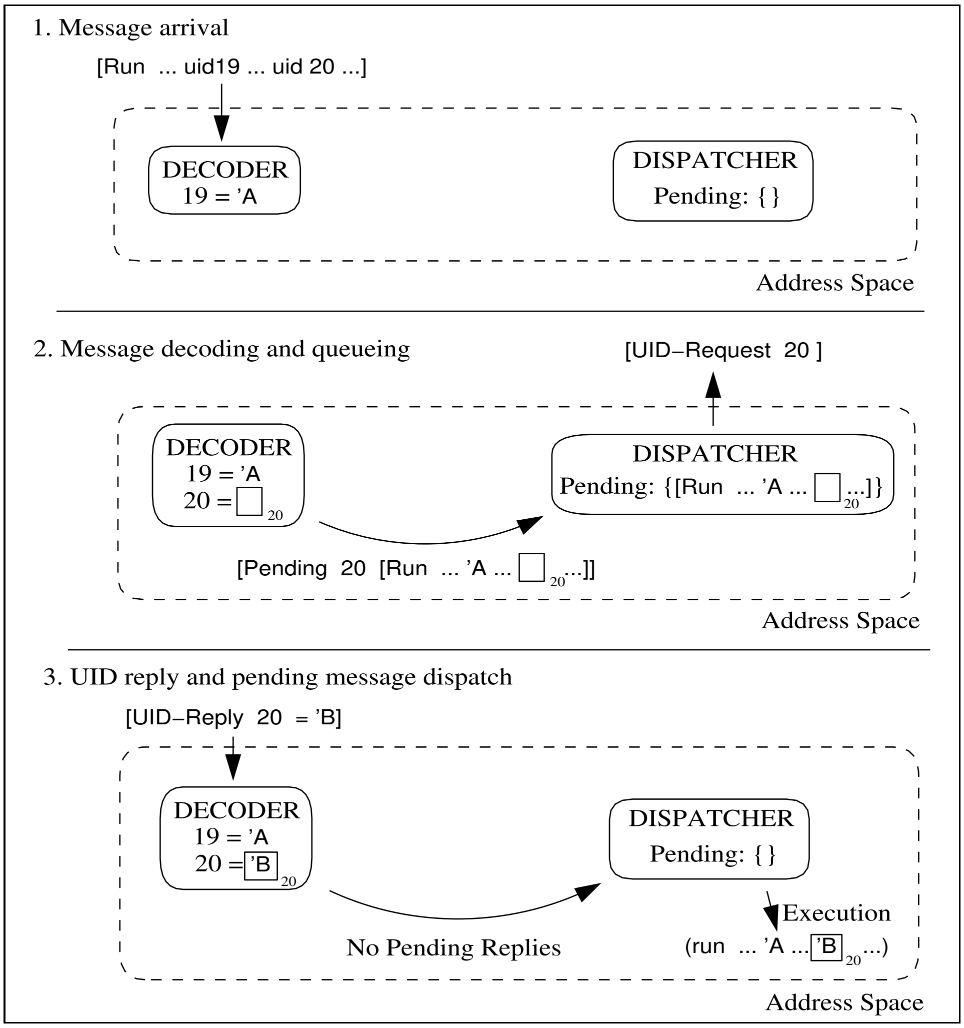


Fig. 12. A run message containing unknown uids received by an address space (1) causes the uid to be recorded and the message stored on a pending queue (2). A request for the object associated with the uid is made to the sender. When the reply is received (3), the message is removed from the pending queue and scheduled for execution.

```
(define (fault-more-frames rest-cont-handle . args)
  ((handle-value rest-cont-handle) args)))

(define (run-cont-on-owner rest-cont-handle args)
  (remote-run! (hanlde-creator rest-cont-handle)
    (lambda () ((handle-value rest-cont-handle) args))))
```

Fig. 13. Among other options, a handler which encounters a remote continuation can choose to have more of the continuation's frames faulted over to its address space, or it can choose to have the rest of the continuation executed on the address space in which the continuation was originally created. `Fault-more-frames` implements the first alternative, and `run-cont-on-owner` implements the second.

### 4.3 Continuations

Scheme 48 threads are first-class objects represented as continuations. In order to support long-lived distributed computations, it is important to allow threads to migrate efficiently between different address spaces based on load-balancing and locality criteria. Continuations may refer to a large amount of state. Thus, migrating a thread by simply copying its entire continuation from one address space to another is likely to be too expensive to be practical. To alleviate the overheads in transmitting continuations, we implement a form of computation migration [Hsieh et al. 1993].

Sending a thread's continuation involves sending only its top few frames, together with a handle stored at the base of the sent continuation. The value slot of this handle contains a reference to the rest of the continuation stack. On the receiving end, a thread is spawned that resumes the continuation. During the course of its computation, such a thread might never return to its continuation's base frame; this is particularly likely to be the case if threads are long lived or subject to frequent migration. In the latter case, avoiding the copy of the entire continuation stack can significantly reduce network traffic. Of course, if the top frames of the continuation do refer to a large amount of data, a significant amount of copying may take place when the continuation is first migrated. Since this data is at the top of the stack, it is highly likely that it will be referenced by the continuation when it is resumed by the receiver; this is less likely to be the case for data stored at deeper frames in the continuation stack.

When control does pass through the base of a remote continuation object, an exception is raised. The executing handler has several options. For example, it may choose to execute a `remote-run!` operation in the address space on which the proxy found at the base was created. This operation might fault several more frames of the continuation back to the receiver. When these frames are received, the handler fixes up the existing continuation and resumes the thread. Alternatively, the handler may choose to apply the value of the proxy on the proxy's owner; in this case, the thread migrates back to its original home. Figure 13 illustrates these two alternatives.

```
(define (move-to! aspace)
  (call/cc (lambda (k)
             (remote-run! aspace k #f)
             (terminate-current-thread))))

(define (move-thread! thread aspace)
  (cond ((eq? thread current-thread)
         (move-to! aspace))
        (else
         (remove-from-queue! runnable-threads thread)
         (remote-run! aspace (thread-continuation thread) #f))))
```

Fig. 14. Both `move-to!` and `move-thread!` manipulate continuations. `Move-to!` runs the continuation of the current thread on the target address space; `move-thread!` runs the continuation of its input thread on the target address space. Both procedures terminate the thread on the source address space by either scheduling another thread to run (in the case of `move-to!`) or removing the thread from the runnable thread queue (in the case of `move-thread!`).

## 5. APPLICATIONS AND IMPLEMENTATION TECHNIQUES

### 5.1 User-Level Scheduling

Representing threads in terms of first-class procedures and continuations enables a number of interesting and novel paradigms. One of the most-important advantages of such abstractions is the ability to migrate threads between address spaces using the same techniques used to migrate procedures. A thread scheduler capable of migrating threads between address spaces can be expressed entirely in user code; moving a thread from one address space to another only involves using operations on continuations. Figure 14 defines two procedures to move threads from their current address space to a new one.

When called with a target aspace, `move-to!` does the following:

(1) It captures the continuation of the currently executing thread.

(2) It does a `remote-run!` of this continuation in the target address space. The `remote-run!` operation will create a new thread to run this continuation.

(3) It terminates the thread on the source address space.

`Move-thread!` is similar to `move-to!`, except that it runs the continuation of its input thread on the target address space.

One can envision a number of variations to this basic strategy that provide greater control and flexibility over how and where threads migrate. All of these variations, however, build on the basic flexibility afforded by first-class continuations and procedures. For example, Figure 15 shows the basic structure of a thread scheduler that uses a centralized master to offload tasks. The master records loads on different address spaces. The `offload` procedure queries the master address space to find a lightly loaded address space if the load on its address space exceeds some threshold. The `record-move` procedure records the migration; load information is adjusted appropriately.

In most distributed systems, monitoring the behavior of procedures such as `offload` or `find-aspace` is problematic since little if any support is provided for

```
(define (offload)
  (let ((AS (current-aspace)))
    (when (> (get-load AS *max-threshold*)
      (let ((thread (find-thread-to-migrate runnable-threads))
            (target (remote-apply master find-aspace AS)))
        (remote-apply master record-move AS target thread)
        (move-thread! thread target)))))
```

Fig. 15. The `offload` procedure coordinates with a central master address space to determine a target address space.

such interaction. Programmers must usually hardwire monitoring facilities into these procedures. Kali Scheme gives users much greater flexibility to monitor and debug ongoing computations. For example, a user can send messages to any address space from a read-eval-print loop executing on any node in a configuration. In this particular case, a user can type the following expression at a REPL to print a message if the load on a particular address space exceeds some threshold:

```
> (remote-apply master
     (lambda (source as)
       (if (> (get-load as) *max-threshold*)
           (remote-apply source display "Threshold exceeded")))
     (current-aspace)
     AS)
;;; the ">" represents a REPL prompt.
```

The template associated with the expression, `(lambda (source as) ...)`, is constructed on the address space of the sender and dynamically shipped to the master. There is no constraint requiring the evaluation of the above expression to occur on the same node in which the address space containing the master resides.

Scheduling policies in a distributed system are greatly simplified through the use of proxies. This is especially the case if these policies allow task migration between address spaces. Consider the definition of a distributed task queue. Tasks can be freely created by other tasks and can migrate among address spaces. When a task completes on address space $A$, a new task is scheduled and run from the set of ready tasks on $A$. We define a simple implementation of this abstraction in Figure 16.

The `create-task-dispatcher!` procedure creates a new task queue and spawns a task dispatcher on the specified address space. Consider a call to `add-task!` made by some task executing on address space $A$. The effect of the call is to have a new task enqueued on the task queue associated with $A$. Because task queues are proxies, the enqueue operation is visible only to dequeue operations performed by the `run-task!` thread running on $A$.

When a task completes, the `run-tasks` procedure dequeues a new task and runs it. Proxies simplify the implementation because `add-task!` and `run-task!` need not be aware of the address space on which they are called; when executed on address space $A$, `(proxy-value task-queue)` returns the value of the task queue on $A$. By avoiding the need to pass address spaces explicitly as arguments to these procedures, a high degree of abstraction is preserved. Note also a task's continuation ends with a call to `loop` in `run-tasks`. This call causes a new task to

```
(define task-queue (make-proxy #f))

(define (add-task! task)
  (enqueue! (proxy-value task-queue) task))

(define (run-tasks)
  (let loop ()
    (let ((new-task (dequeue! (proxy-value task-queue))))
      (new-task))
    (loop)))

(define (create-task-dispatcher! aspace initial-task)
  (remote-apply aspace
    (lambda ()
      (set-proxy-value! task-queue (make-queue))
      (add-task! initial-task)
      (spawn (run-tasks)))))
```

Fig. 16. An implementation of a simple distributed task queue. This implementation assumes tasks are nonpreemptible. The `dequeue!` procedure is assumed to block if its argument queue is empty.

be dequeued off a queue accessed via the `task-queue` proxy. As a result, migrating tasks to other address spaces requires no special care.

### 5.2 Client-Server Applications

The ability to download code dynamically can be used to reduce communication overhead by tailoring the communication protocol to the task at hand. Client-server applications are a good example where dynamic modification of an initial communication protocol is especially important. In most systems, client-server protocols are fixed. This design decision is an expensive one if clients and servers are not tightly coupled, if response times on client or server machines are not negligible, or if message latency is high.

As an example, consider a graphics application that displays a map of the world. The input to the program is an atlas represented as a list of countries. Each country has a name (a string) and a list of closed curves that define the country's boundaries. The purpose of the application is to:

(1) Open a window.
(2) Display the boundaries of all countries.
(3) Track the mouse. If the mouse in a country, the country is colored, and its name is displayed.
(4) Monitor keystrokes. When the return key is struck, return the name of the country on which the mouse is placed (if any) and exit.

For our purposes, the application becomes interesting when it is run on a client machine different from the server machine controlling the display. On tightly coupled networks, the latency cost in waiting for every mouse event to be sent to the client for translation may not be severe enough to warrant special attention. However, if the client and server are on a wide-area network, or if network bandwidth

```
(define (pick-a-country atlas)
  (let* ((point-transformer (make-transform (window-bounding-box)
                                            (atlas->bounding-box atlas)))
         (t-atlas (transform-atlas point-transformer atlas))
         (mouse->countries (make-mouse->countries t-atlas))
         ...)
    (draw-atlas! t-atlas)
    (let loop ((selected-countries '()))
      (let ((event (get-event)))
        (case (event->type event)
          ((motion)
           (let ((new-countries (mouse->countries
                                 (motion->position event))))
             (cond ((not (equal? selected-countries new-countries))
                    (for-each unhighlight! selected-countries)
                    (for-each highlight! new-countries))
             (loop new-countries))))
          ((key-press)
           (cond ((eq? 'return (key-press->key-name event))
                  (destroy-window!)
                  (map country->name selected-countries))
                 (else (loop selected-countries))))
          (else
           (loop selected-countries))))))
```

Fig. 17. An outline for a graphical atlas program.

and latency are not sufficient, communication demands placed by this application will be severe enough to justify a more-sophisticated work partition between the server and the client.

Ignoring remote communication, we provide a simple outline of the application in Figure 17. The `make-transform` procedure returns a procedure that implements a transformation from the bounding box of the atlas to the bounding box of the window on which the atlas is to be displayed. Thus, `t-atlas` is the input atlas with input coordinates transformed to screen coordinates. The procedure `make-mouse->countries` returns a procedure `mouse->countries` closed over a large table that associates a screen point with the list of countries in the atlas which overlap that point; given a mouse position, `mouse->countries` returns the list of countries that the mouse lies within.

In the code shown, every mouse event involves communication between the server controlling the display and the application containing the atlas and `mouse->countries` procedure. If the server runs on `aspace`, evaluating

```
(remote-apply aspace pick-a-country atlas)
```

will cause the application to run on the same machine as the server, returning the countries on which mouse resides when the user hits the return key back to the client machine. Alternatively, by rewriting the procedure slightly we can transmit less data (see Figure 18). In this version, the client computes the coordinates of countries in the atlas to conform to screen coordinates, but the association between cursor positions and countries in the atlas is computed by the server. Downloading code in this way simplifies the protocol and allows the dynamic partitioning of

```
(define (pick-a-country atlas server-aspace)
  (let* ((point-transformer (make-transform (window-bounding-box)
                                            (atlas->bounding-box atlas)))
         (t-atlas (transform-atlas point-transformer atlas)))
    (remote-apply server-aspace
      (lambda ()
        (let ((mouse->countries (make-mouse->countries t-atlas))
              ...)
          ...)))))
```

Fig. 18. We can offload functionality from the client to the server by using address spaces and remote application.

```
(define (make-mouse->countries t-atlas)
  (let ((table-handle (make-handle
                        (make-mouse->countries-table t-atlas)))
        (mirror (make-mirror-table)))
    (lambda (mouse-position)
      (let ((countries (mirror-table-lookup mirror mouse-position)))
        (if (unknown? countries)
            (let ((countries (remote-apply (proxy-creator table-handle)
                               (lambda ()
                                 (lookup-mouse->countries
                                   (handle-value table-handle)
                                   mouse-position)))))
              (add-mirror-table! mirror mouse-position countries)
              countries)
            countries)))))
```

Fig. 19. By incrementally building on the server a mirror table that associates screen coordinates with countries, we can further reduce communication costs. Table entries are retrieved from the client only when the mouse moves over a portion of the atlas whose coordinates are not locally available on the display server. We assume a table representation that provides efficient access to sparse data.

work between machines. Sun's NeWS [Sun Microsystems 1990] permits code to be transmitted in a similar way, with applications downloading Postscript programs to the server. The NeWS's protocol is much lower level than ours however, since programs must be passed as source text or as token sequences. It has no automatic encoding of data structures or significant support for distributed programming and uses a much lower level programming language.

Finally, by making the implementation of `make-mouse->countries` aware of address spaces, we can reduce communication costs still further. If the procedure returned by `make-mouse->countries` wrapped a handle around the table it builds, the server can incrementally construct a mirror copy of this table on demand. The code to effect this incremental faulting of the mouse/country table from the client to the server is outlined in Figure 19.

```
(define (make-dna-workers worker-aspaces)
  (let ((db-proxies (map (lambda (aspace)
                           (remote-apply aspace make-proxy '()))
                         worker-aspaces))
     (values (make-add! db-proxies)
             (make-find-nearest db-proxies))))
```

Fig. 20. `Db-proxies` is a list of proxies. The value of the $i$th element contains the portion of the database seen by the $i$th worker.

### 5.3 Parallelism

As a final example, we consider the implementation of a DNA sequence comparison algorithm [Carriero and Gelernter 1990]. The algorithm takes as input a target string and a database and initiates a parallel search for the string in the database, returning as its result the "closeness" of the element in the database that bears the greatest similarity to the target. A natural implementation strategy is to use a master-worker arrangement; the master creates a number of worker threads with each worker responsible for comparing the target to a specific portion of the database. When a thread finds its local best match, it notifies the master and terminates. The master completes when all threads complete, returning the global best string found. This kind of application is well suited for implementation in Kali Scheme and benefits from the liberal use of higher-order procedures.

Associated with each worker is a proxy owned by the address space where the worker will live. This proxy will contain the portion of the database that is to be examined by this worker. In addition, we require a procedure to add new elements to a worker's portion of the database. This procedure along with the search routine used by a worker is given by the procedure shown in Figure 20.

The `make-add!` procedure simply keeps track of the total length of all the entries in each worker's portion of the data base. When given a new item to add, it finds the worker with the smallest local data base and adds the entry to that workers proxy. Given the proxy, and the value to add to that database, the addition is accomplished by evaluating the following expression:

```
(remote-apply (proxy-creator proxy)
  (lambda ()
     (set-proxy-value!
       proxy
       (cons dna (proxy-value proxy)))))
```

`Remote-apply` (versus `remote-run!`) is used here simply for the synchronization it provides. Note that `dna` is implicitly copied to the worker address space because it occurs free in the procedure evaluated by `remote-apply`.

`Make-find-nearest` returns a procedure which, when called on a string, returns the distance of the closest item in the database to this string. To do this, it spawns one thread per worker. Each thread will find the closest entry in that worker's database (by calling `local-find-nearest`) and will then update a location which has the closest item found so far. If it is the last worker to finish, it writes the result in a condition variable which is then read by the top-level thread. This is

```
(define (make-find-nearest db-proxies)
  (let ((nworkers (length db-proxies)))
    (lambda (dna-goal)
      (let ((lock (make-lock))
            (left nworkers)
            (so-far infinitely-far-entry)
            (final-result (make-condvar)))
        (for-each
          (lambda (db-proxy)
            (spawn (lambda ()
                     (let ((local-best (local-find-nearest
                                         dna-goal
                                         db-proxy)))
                       (with-lock lock
                         (set! so-far (closest-entry so-far local-best))
                         (set! left (- left 1))
                         (if (zero? left)
                             (condvar-set! final-result so-far))))))))
           db-proxies)
          (condvar-ref final-result))))))
```

Fig. 21. The master task for a DNA database search program.

```
(define (local-find-nearest dna-goal db-proxy)
  (remote-apply (proxy-creator db-proxy)
    (lambda ()
      (let loop ((best-entry infinitely-far-entry)
                 (entries (proxy-value db-proxy)))
        (if (null? entries)
             best-entry
             (loop (closest-entry best-entry
                                  (closeness dna-goal (car entries)))
                   (cdr entries)))))))
```

Fig. 22. Worker threads execute `local-find-nearest` to find the closest match on their portion of the database.

the heart of the master task and is shown in Figure 21.

`Local-find-nearest` , given the DNA string we are searching for and the proxy holding a workers local database, simply executes a `remote-apply` operation targeted for that worker's address space; the procedure spawned folds over the local database to find the closest item; its definition is given in Figure 22.

## 6. GARBAGE COLLECTION

Local garbage collection (GC) can proceed asynchronously in each address space, as long as proxies and their local values are preserved. Proxies cannot be collected locally since any proxy that has been sent to or received from another address space is potentially accessible from other address spaces via `remote-run!` operations. In this section we will describe two nonlocal GC algorithms for proxies, an asynchronous one that cannot collect cycles, and a synchronous one that can.

## 6.1 Asynchronous Garbage Collection

The asynchronous GC algorithm used in this system is designed with the following assumptions in mind:

(1) Remote references are likely to be ephemeral; thus, creating and destroying them should be cheap and entail minimal bookkeeping overhead.
(2) Message communication is costly relative to computation; thus, garbage collection should minimize the use of synchronization messages even if this leads to a less-aggressive collection strategy.
(3) Messages are delivered in FIFO order between any two processors, but there is no bound on communication latency. There is no assumed arrival order on messages sent to different address space by the same sender.

The first assumption is a consequence of the functional programming style encouraged by Scheme. Because mutation is rare, most objects tend to be short lived; objects tend to be allocated and deallocated much more frequently than they tend to be mutated [Reinhold 1994]. Procedures also tend to be small and lightweight. Consequently, in any distributed system based on Scheme, remote references will tend to have short lifetimes. The latter two assumptions derive from constraints imposed by commercially available (low-end) network technology and protocols that we have used in testing our prototype.

The GC algorithm uses two one-bit flags in each proxy. The *enroute* flag indicates that the proxy has been included in an outgoing message since its creator's last asynchronous proxy collection. The *referenced* flag is set when a proxy is created or when it is received in a message. The flag is cleared when a local collection reveals that there is no local reference to the proxy. Proxies with no local value whose enroute flag is clear may be collected by an address space at any time.

The procedure used by an address space, $A$, to garbage collect proxies that it has created is as follows:

(1) $A$ notifies all address spaces that it is now garbage collecting its proxies.
(2) When address space $B$ receives $A$'s GC notification, it sends to $A$ the set of $A$'s proxies currently extant on $B$ with either flag set and clears the enroute flags on these proxies.
(3) $A$ can reclaim any proxies that have no local references and were not on any of the lists of proxies returned by other address spaces. $A$ also notifies $B$ of all proxies it can reclaim; any proxy which $B$ could not originally collect because they had local values can now be collected if they are included in the list returned by $A$.

Figure 23 gives a graphical depiction of the algorithm. This algorithm is simple and asynchronous. It has two fundamental flaws however. The first is that it cannot to collect cycles of proxies; we remedy this by defining a synchronous variant of this collector. The more-serious problem is that the algorithm can potentially reclaim proxies that are still referenced. Repairing this lack of safety requires making the algorithm weakly synchronous.

The following sequence of events illustrates the problem with the algorithm (see Figure 24):

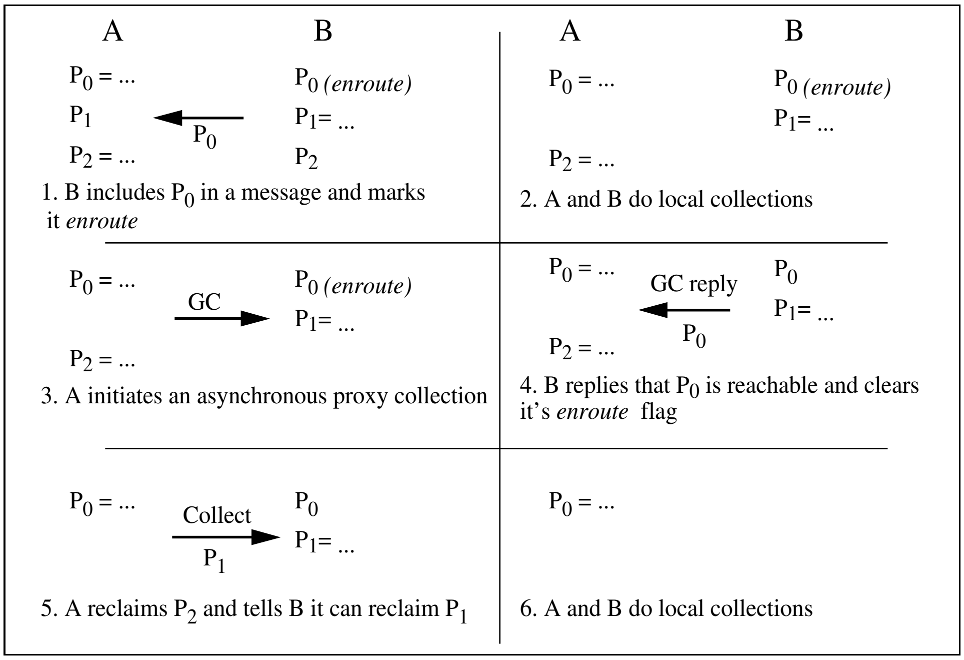


Fig. 23. Asynchronous collection of proxies. All of the proxies shown were created by $A$, and none are referenced locally. Thus, all proxies can be garbage collected. Proxies that have local values are depicted as "P = ..." in the figure.

(1) $A$ creates proxy $p$ and sends it to $B$.

(2) $B$ sends $p$ to $C$, setting $p$'s enroute flag on $B$.

(3) $A$ starts its GC procedure, sending messages to $B$ and $C$.

(4) $B$ reports to $A$ that it has $p$, clears its enroute flag, and reclaims its local copy. $C$ reports that it has none of $A$'s proxies ($p$ is still in transit from $B$).

(5) $A$ finishes its collection and does not reclaim $p$, because $B$ reported having a reference to it.

(6) $A$ performs a second garbage collection. This time neither $B$ nor $C$ report having $p$, so $A$ reclaims it.

(7) The message containing $p$ finally arrives at $C$, which now has a reference to a proxy whose value no longer exists.

The critical observation here is that message transmission is not atomic, and there is no global time-ordering on message events. Thus, while $A$ and $B$ may have consistent images of shared data, consistency does not extend to include shared objects held by $C$ or that are in transit from $B$ to $C$. To remedy these problems, we define a global "tick" thread that creates a coarse-grain time ordering on message transmission. The behavior of this thread guarantees that all messages in transit at the time the tick thread last visited an address space will have been received when the tick next arrives. We assume some linear ordering on address spaces. When running on address space $A$, the tick thread sends a message (via `remote-apply` ) to

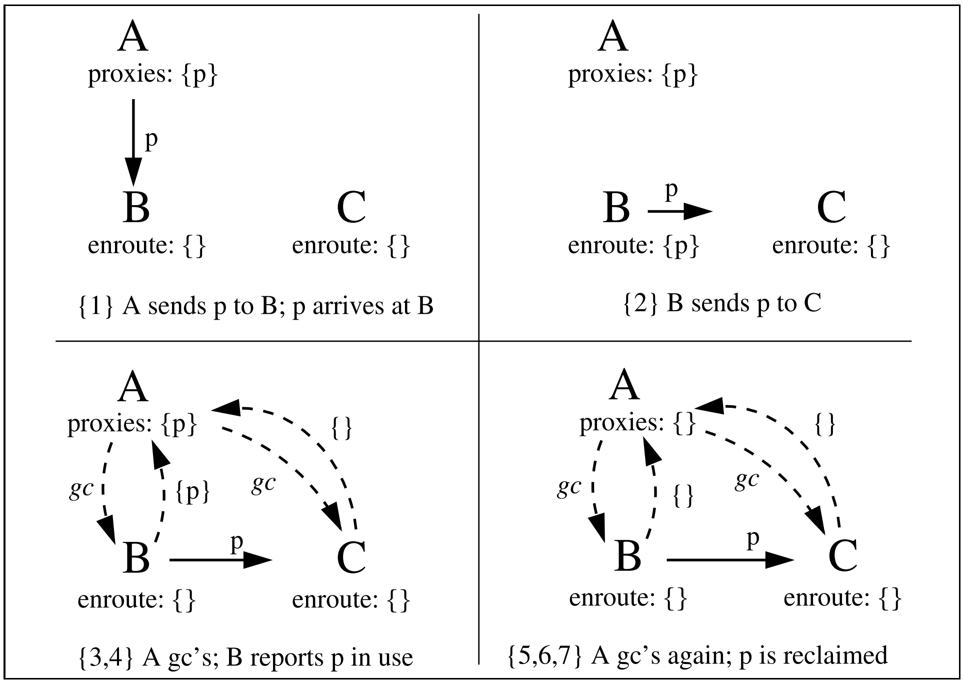


Fig. 24. Because we impose no constraints on communication latency, a simple asynchronous GC strategy may erroneously collect live data associated with an accessible proxy. The numbers in parentheses refer to the corresponding steps in the text.

every address space greater than $A$ in this ordering; this message simply executes the null procedure. When all messages have been acknowledged from all address spaces, the tick thread moves from $A$ over to the next address space in the ordering. When all address spaces have been visited, it is guaranteed that any message in transit before the last iteration of the tick has reached its destination. By waiting for one full tick (two consecutive arrivals of the tick thread) between garbage collections, we can ensure that every proxy that is enroute during one garbage collection will have arrived at its destination in time for the next collection.

This algorithm is simple because it is highly local and relies only on our assumption that messages are delivered in FIFO order on each channel. One can imagine a more-aggressive collection strategy that reclaims dead, remote references more eagerly, but we suspect such schemes are unlikely to be as simple and unobtrusive as this one.

### 6.2 Collecting Cycles

The asynchronous collector is unable to collect cycles of proxies. If proxy $p$, owned by address space $A$, has as its value proxy $q$, owned by $B$, which in turn has $p$ as its value, $A$ will always prevent $B$ from reclaiming $q$ and vice versa. To prevent this scenario, the proxies in a cycle need to be reclaimed simultaneously.

The algorithm we use for handling cycles is as follows:

(1) All address spaces stop computing.

(2) Each address space $A$ uses its local garbage collector to determine which proxies have local references and determines for each proxy $P$ the set of proxies reachable from $P$'s value in $A$. These sets can be interpreted as a graph, with an edge indicating that one proxy's value is reachable from another's.

(3) The address spaces cooperatively walk the graph of reachable proxies, starting from the locally referenced ones, marking each reachable proxy.

(4) All unmarked proxies are reclaimed.

In the worst case, this algorithm requires time linear in the number of proxies. In practice we expect that most applications will not create cycles in the proxy graph and that, for those that do, the vast majority of proxies will still be reclaimed by the asynchronous algorithm.

### 6.3 Comparision to Other Distributed Collectors

Unlike reference-counting collection strategies [Lang et al. 1992], our asynchronous collector requires only two bits to reclaim proxies. Our approach bears greater similarity to distributed marking schemes that use *bulldozing* messages [Kamada et al. 1994; Venkatasubramanian et al. 1992] to confirm message arrival. Unlike these other schemes, the tick process used to provide the effect of bulldozing is itself asynchronous with respect to user processes, requires no extra acknowledgment signals, and is not centralized. The cyclic extension is synchronous, but again its complexity is bounded only by the number of proxies, not by the number of messages or proxies sent.

The garbage collection algorithm for Network Objects [Birrell et al. 1993] shares some similarity with ours. Both systems permit proxies to be forwarded by address spaces other than the owner, and both systems are weakly synchronous insofar as local garbage collections in one address space can occur without requiring global synchronization. However, there are also several important differences. In part, these differences reflect differences in the two systems' capabilities. For example, unlike Kali Scheme, Network Objects was designed to tolerate node failure. The network objects collector uses a reference-counting scheme in which the owner of a shared object $O$ maintains a list $L$ of all processes that have a proxy or surrogate to $O$; clients which no longer refer to a proxy notify the owner. The reference-counting approach also effectively prohibits the collector from reclaiming cycles. In Kali Scheme two simple flags are used to reclaim proxies; no reference lists are maintained. The owner of a proxy is the only entity that can initiate the proxy's collection. More significantly, the network objects collector assumes a tight coupling between message sends and proxy (or surrogate) collection. To avoid race conditions of the kind outlined above, the Network Objects implementation uses acknowledgments to record network handles that are enroute from one address space to another; during transmission, such proxies are never collected by the sender. Because of the system's RPC semantics, acknowledgments can piggy-back on method return calls, although special acknowledgment signals are still necessary when a proxy is returned as a result of an RPC call. Kali Scheme's collector requires no acknowledgments from proxy receivers on `remote-run!` operations.

Table I. Baseline Times

| | R4000 | R3000 |
|---|---|---|
| Times to execute in milliseconds | | |
| Remote-run! | 2.3 | 3.9 |
| Remote-apply | 4.6 | 8.1 |
| Time to send in $\mu$-seconds | | |
| Proxy | 30.5 | 57.5 |
| Integer arg | 27.1 | 44.1 |
| Vector | 105 | 131 |
| Procedure | 41.5 | 172 |
| Thread | 170 | 576 |

## 7. PERFORMANCE

Because the sequential core of our system compiles to byte codes, the overall performance of the system will obviously be poorer than a native code implementation. To validate the utility of our language abstractions, the byte-code implementation appears to be a reasonable compromise between a very inefficient, but simple, source-level interpreter and an efficient, but less portable, native-code compiler. Since our primary goal was investigating the applicability of higher-order language features to express interesting abstractions for distributed computing, we do not view the lack of native-code performance as critical to assessing the system's utility. In the presence of a native-code compiler, we would expect to see a factor of 10-15 fold reduction in message interpretation overhead.

Since the message decoder is implemented in the virtual machine, the baseline times for message transmission are nonetheless very good despite the fact that all operations require byte-code interpretation. Table I gives baseline times for the system. The first column gives measurements taken on two MIPS R4000 100MHz processors connected via a 10Mb/s Ethernet. The second column provides times for the benchmarks on 33MHz R3000 machines also connected via a 10Mb/s Ethernet. Note that the cost of sending a TCP/IP packet in C from user space to user space takes roughly 1.9 milliseconds on our network. This number was determined by dividing by two the time required to send a packet and then receive a reply.

As another baseline comparison, in Scheme 48, it takes 4.09$\mu$-seconds to execute the null procedure on an R4000 and 8.02$\mu$-seconds to execute it on an R3000. It takes 378 $\mu$-seconds on an R4000 to spawn a thread that calls the null procedure, to schedule it, apply its thunk, and then return; it takes 1060 $\mu$-seconds on an R3000 to do the same operation.

The first two rows measure the cost of a `remote-run!` and `remote-apply` operation of a null procedure; `remote-apply` measures round-trip time. Using the R4000 times for a `remote-run!` we see that, after subtracting the overhead of TCP/IP (1.9 milliseconds), 400 $\mu$-seconds are spent in Scheme 48. Of this time, 378 $\mu$-seconds are involved with thread creation, context switching, and synchronization, indicating that only roughly 12 $\mu$-seconds are spent in the transport layer for decoding and dispatching. As we would expect, the overheads for executing a `remote-run!` are dominated foremost by TCP/IP costs and secondly by thread management overheads. The transport layer contributes only a small overhead compared to these other costs. This is because the transport layer is implemented within the

Scheme 48 virtual machine which is compiled as C code. Note that in an implementation which moved thread functionality into the virtual machine, TCP/IP costs would be even more dominant.

The next four rows measure the cost of sending a proxy, an integer argument, a vector of 10 elements, a procedure with an empty environment, and a minimal thread. Each of these rows measures overheads above a simple `remote-run!` operation. The `remote-apply` time is, as one would expect, roughly twice the cost of a `remote-run!`. The cost of sending a proxy is roughly the same as sending a simple integer argument; the extra overhead is due to administrative costs to record the receipt of the proxy in appropriate tables. The extra cost of sending a thread over that of a procedure is due in large part to the need to migrate the thread's dynamic environment, i.e., the top few frames of its continuation.

## 8. RELATED WORK AND CONCLUSIONS

The lightweight, distributed communication facility described here differs in important respects from RPC mechanisms found in many distributed languages [Birrell and Nelson 1984; Schroder and Burrows 1990]. Since these languages are typically first order, the issue of sending dynamically instantiated closures is usually not considered. Secondly, the marshaling code used in our system permits any Scheme object (including references, continuations, top-level new definitions, etc.) to be sent in a message; there is no constraint on the kinds of objects that can be sent.

Our notion of *proxies* is a generalization of the "network objects" described in Birrell et al. [1994]. Unlike network objects, the objects encapsulated within a proxy are not consistent across address spaces. However, since a proxy has a unique owner, it can be used to express the behavior of a shared data object. More importantly, the design and implementation of network objects is not geared with first-class procedures and continuations in mind; in contrast, many of the applications we have considered, and much of the implementation itself, liberally use higher-order structures.

Proxies are also not associated with a given set of methods; since Scheme is latently typed, there is also no type enforcement on proxies, nor are there any explicit stub modules to generate marshaling code for different Scheme types. Because of Scheme's latent typing discipline, type errors are detected at runtime. Thus, consider a `remote-run!` operation evaluated on address space $A$ that spawns a thread $T$ on address space $B$. The application of the closure associated with $T$ may raise a runtime exception on $B$, even though it was defined on $A$. Type errors are a common example of such exceptions that could be flagged at compile-time. There has been much progress in building type systems and optimizers for Scheme that catch many potential type errors statically [Jagannathan and Wright 1995; Wright and Cartwright 1994]. We believe incorporating these facilities into Kali Scheme can significantly alleviate debugging overheads that would otherwise be incurred.

Obliq [Cardelli 1995] is a higher-order distributed language built on top of network objects. Obliq allows immutable data to be freely transmitted but generates network references for mutable structures. Obliq also does not support processor-relative addressing using proxies. In contrast, our implementation makes no implicit assumptions on how objects are to be transmitted; any Scheme object (including a thread or a continuation) can be explicitly copied or referenced via proxy. Obliq also does not support first-class continuations, although the availability of first-class functions makes it possible to program in continuation-passing style. Finally, Obliq is implemented as a source-level interpreter. In contrast, we do rely heavily on being able to send (partial) continuation objects across different address spaces, and our implementation is intended for compiled programs.

Java [Sun Microsystems, Inc. 1995] is a byte-coded system intended to work in heterogeneous distributed environments. Like Kali Scheme, Java allows byte-codes to be transmitted across nodes, but because the sequential core is based on C++, it does not support higher-order abstractions such as procedures, continuations, and first-class threads. The language also does not define any sharing abstractions similar to proxies.

CML [Reppy 1991] is a concurrent dialect of ML that supports first-class events, channels, and preemptible threads. Our system shares some common features with CML insofar as both rely heavily on first-class continuations and procedures in their implementation. However, CML does not provide explicit support for distributed computation; thus, it does not include abstractions for building remote references or address spaces. It also does not consider distributed garbage collection across disjoint address spaces. Facile [Giacalone et al. 1990] is an extension of ML that is intended to operate in a distributed environment. The concurrency model used in Facile is a generalization of CCS [Milner 1989]. Consequently, distributed computing in Facile is based on synchronous communication using channels; the language does not have analogues to address spaces, proxies, system support for asynchronous message passing, or the other primitives described here.

Piranha [Carriero et al. 1995] is an implementation of C.Linda that runs on networks of workstations. Unlike our system, the lack of first-class continuation objects makes it problematic for Piranha to adapt gracefully to changing network loads. A Piranha process that *retreats* from a node must either discard all work it has so far performed on its current task or require the programmer to construct manually the continuation object that is to be executed upon its resumption. Reification of a task state into a continuation is problematic in the C context; in contrast, because threads are implemented in terms of continuations, migrating a thread (i.e., retreating from a processor) is an easily specified operation in our system. In addition, because Piranha is a tuple-space implementation, the fundamental communication abstraction (shared data expressed in terms of tuples) is conceptually distinct from the basic proxy abstraction defined here. Although one can presumably be implemented in terms of the other, we would argue that in our particular context, proxies provide a lighter-weight and more-flexible communication abstraction. Other parallel systems that operate over separate address spaces and which support message-based communication (e.g., PVM [Sunderam 1990]) have similar differences.

Most parallel dialects of higher-order languages such as Scheme or ML (e.g., Jagannathan and Philbin [1992] and Morrisett and Tolmach [1993]) execute in a single address space. These systems do not provide primitives to allow threads explicitly to communicate across disjoint address spaces. Consequently, their semantics, implementation, and targeted applications differ in many important respects from the ones described here. Split-C [Culler et al. 1993] is a parallel extension of C that provides a global address space. The design of Split-C shares some commonality with ours — allowing programmers a significant amount of control over communication costs. However, because it is an extension of C, the programming model and paradigms it supports differ in obvious ways from a distributed system based on first-class procedures, first-class continuations, and incremental distributed linking of code objects.

Process migration [Powell and Miller 1983] and computation migration [Hsieh et al. 1993] are two approaches to moving threads in distributed environments. Our implementation shares much in common with computation migration insofar as continuations are migrated lazily. Unlike Hsieh et al. [1993], our continuation migration protocols are integrated within the Scheme 48 exception-handling facility — handlers on different nodes can choose to implement different functionality when control passes below the sent continuation base. For example, an error raised by a thread can be treated by either migrating the thread back to its original home or faulting the error handler over to the thread's current address space.

Scheme's support for first-class procedures and continuations make it an ideal platform in which to explore new paradigms and idioms for distributed computing. The ability to express new abstractions in terms of those already available in the language greatly simplifies the implementation. More importantly, it allows users to build nontrivial refinements and extensions without reengineering the system from scratch. We conclude that high-level data and control abstractions have much to offer in domains such as distributed computing and that there are a number of important benefits in using these abstractions to express distributed applications that merit continued investigation.

A copy of the system with sources can be obtained through URL http://www.neci.nj.nec.com/PLS/kali.html.

## ACKNOWLEDGMENTS

Thanks to the anonymous referees, Luca Cardelli, and John Ellis for their many useful comments and suggestions.

## REFERENCES

ANDREWS, G. 1991. *Concurrent Programming: Principles and Practice*. Benjamin-Cummings, Menlo Park, Calif.

ARVIND, NIKHIL, R., AND PINGALI, K. 1989. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Sys. 11,* 4, 598–632.

BIRRELL, A., EVERS, D., NELSON, G., OWICKI, S., AND WOBBER, E. 1993. Distributed garbage collection for network objects. Tech. Rep. Digital SRC Research Rep. 116, Digitial Equipment Corp., Palo Alto, Calif.

BIRRELL, A., NELSON, G., OWICKI, S., AND WOBBER, E. 1994. Network objects. Tech. Rep. Digital SRC Research Rep. 115, Digitial Equipment Corp., Palo Alto, Calif.

BIRRELL, A. D. AND NELSON, B. 1984. Implementing remote procedure call. *ACM Trans. Comput. Syst. 2,* 1, 39–59.

CARDELLI, L. 1995. A language with distributed scope. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*. ACM, New York, 286–298.

CARRIERO, N. AND GELERNTER, D. 1990. *How to Write Parallel Programs: A Guide to the Perplexed*. MIT Press, Cambridge, Mass.

CARRIERO, N., GELERNTER, D., JOURDENAIS, M., AND KAMINSKY, D. 1995. Piranha scheduling: Strategies and their implementation. *Int. J. Parallel Program. 23,* 1, 5–35.

CLINGER, W. AND REES, J. 1991. Revised[4] report on the algorithmic language Scheme. *ACM Lisp Pointers 4,* 3 (July), 1–55.

COX, A., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., AND ZWAENEPOEL, W. 1994. Software versus hardware shared-memory implementation: A case study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture.* IEEE, New York, 106–119.

CULLER, D., DUSSEAU, A., SETH COPEN, G., KRISHNAMURTHY, A., LUMETTA, S., THORSTEN VON, E., AND YELICK, K. 1993. Parallel programming in Split-C. In *Proceedings of the 1993 ACM Symposium on Supercomputing.* ACM, New York.

CURTIS, P. 1992. Mudding: Social phenonmena in text-based virtual realities. Tech. Rep. CSL-92-4, Xerox Palo Alto Research Center, Palo Alto, Calif.

GIACALONE, A., MISHRA, P., AND PRASAD, S. 1990. Facile: A symmetric integration of concurrent and functional programming. *Int. J. Parallel Program. 18,* 2.

HAYNES, C. AND FRIEDMAN, D. 1987. Embedding continuations in procedural objects. *ACM Trans. Program. Lang. Syst. 9,* 4, 582–598.

HSIEH, W., WANG, P., AND WEIHL, W. 1993. Computation migration: Enhancing locality for distributed-memory parallel systems. In *The 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, New York, 239–249.

JAGANNATHAN, S. AND PHILBIN, J. 1992. A foundation for an efficient multi-threaded scheme system. In *Proceedings of the 1992 Conference on Lisp and Functional Programming.* ACM, New York, 345–357.

JAGANNATHAN, S. AND WRIGHT, A. 1995. Effective flow-analysis for avoiding runtime checks. In *Proceedings of the International Static Analysis Symposium*, Lecture Notes in Computer Science, vol. 983. Springer-Verlag, Berlin, 207–224.

KAMADA, T., MATSUOKA, S., AND YONEZAWA, A. 1994. Efficient parallel garbage collection on massively parallel computers. In *Proceedings of the 1994 IEEE Supercomputing Conference.* IEEE, New York, 79–88.

KELSEY, R. 1992. Tail-recursive stack disciplines for an interpreter. Tech. Rep. NU-CCS-93-03, Northeastern Univ., College of Computer Science, Boston, Mass.

KELSEY, R. AND REES, J. 1994. A tractable scheme implementation. *Lisp Symbol. Comput. 7,* 2, 315–335.

KRANZ, D., JOHNSON, K., AGARWAL, A., KUBIATOWICZ, J., AND LIM, B.-H. 1993. Integrating message-passing and shared-memory: Early experience. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, New York, 54–63.

KRANZ, D., KELSEY, R., REES, J., HUDAK, P., PHILBIN, J., AND ADAMS, N. 1986. ORBIT: An optimizing compiler for Scheme. *ACM SIGPLAN Not. 21,* 7 (July), 219–233.

LANG, B., QUEINNEC, C., AND PIQUER, J. 1992. Garbage collecting the world. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages.* ACM, New York, 39–50.

LI, K. AND HUDAK, P. 1989. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst. 7,* 4, 321–359.

MILNER, R. 1989. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, N.J.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML.* MIT Press, Cambridge, Mass.

MORRISETT, J. G. AND TOLMACH, A. 1993. Procs and Locks: A portable multiprocessing platform for Standard ML of New Jersey. In *The 4th ACM Symposium on Principles and Practice of Parallel Programming.* ACM, New York, 198–207.

MULLENDER, S., Ed. 1993. *Distributed Systems.* Addison-Welsey, Reading, Mass.

POWELL, M. AND MILLER, B. 1983. Process migration in demos/mp. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles.* ACM, New York, 110–119.

REINHOLD, M. 1994. Cache performance of garbage-collected programs. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation.* ACM, New York, 206–217.

REPPY, J. 1991. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*. ACM, New York, 293–306.

SCHRODER, M. AND BURROWS, M. 1990. Performance of firefly rpc. *ACM Trans. Comput. Syst. 8,* 1, 1–17.

Sun Microsystems 1990. *NeWS 2.1 Programmer's Guide*. Sun Microsystems, Mountain View, Calif.

Sun Microsystems, Inc. 1995. *The Java Language Specification*. Sun Microsystems, Inc., Mountain View, Calif.

SUNDERAM, V. 1990. Pvm: A framework for parallel distributed computing. *Concurrency: Pract. Exper. 2,* 4.

VENKATASUBRAMANIAN, N., AGHA, G., AND TALCOTT, C. 1992. Scalable distributed garbage collection for systems of active objects. In *Memory Management*, Lecture Notes in Computer Science, vol. 637. Springer-Verlag, Berlin, 134–147.

WRIGHT, A. AND CARTWRIGHT, R. 1994. A practical soft type system for scheme. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*. ACM, New York, 250–262.

Received April 1995; revised July 1995; accepted September 1995