Concurrency Oriented Programming in Termite Scheme

Guillaume Germain Marc Feeley Stefan Monnier
Université de Montréal
{germaing, feeley, monnier}@iro.umontreal.ca

Abstract

Termite Scheme is a variant of Scheme intended for distributed computing. It offers a simple and powerful concurrency model, inspired by the Erlang programming language, which is based on a message-passing model of concurrency.

Our system is well suited for building custom protocols and abstractions for distributed computation. Its open network model allows for the building of non-centralized distributed applications. The possibility of failure is reflected in the model, and ways to handle failure are available in the language. We exploit the existence of first class continuations in order to allow the expression of high-level concepts such as process migration.

We describe the Termite model and its implications, how it compares to Erlang, and describe sample applications built with Termite. We conclude with a discussion of the current implementation and its performance.

General Terms Distributed computing in Scheme

Keywords Distributed computing, Scheme, Lisp, Erlang, Continuations

1. Introduction

There is a great need for the development of widely distributed applications. These applications are found under various forms: stock exchange, databases, email, web pages, newsgroups, chat rooms, games, telephony, file swapping, etc. All distributed applications share the property of consisting of a set of processes executing concurrently on different computers and communicating in order to exchange data and coordinate their activities. The possibility of failure is an unavoidable reality in this setting due to the unreliability of networks and computer hardware.

Building a distributed application is a daunting task. It requires delicate low-level programming to connect to remote hosts, send them messages and receive messages from them, while properly catching the various possible failures. Then it requires tedious encoding and decoding of data to send them on the wire. And finally it requires designing and implementing on top of it its own application-level protocol, complete with the interactions between the high-level protocol and the low-level failures. Lots and lots of bug opportunities and security holes in perspective.

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

Termite aims to make this much easier by doing all the low-level work for you and by leveraging Scheme's powerful abstraction tools to make it possible to concentrate just on the part of the design of the high-level protocol which is specific to your application.

More specifically, instead of having to repeat all this work every time, Termite offers a simple yet high-level concurrency model on which reliable distributed applications can be built. As such it provides functionality which is often called middleware. As macros abstract over syntax, closures abstract over data, and continuations abstract over control, the concurrency model of Termite aims to provide the capability of abstracting over distributed computations.

The Termite language itself, like Scheme, was kept as powerful and simple as possible (but no simpler), to provide simple orthogonal building blocks that we can then combine in powerful ways. Compared to Erlang, the main additions are two building blocks: macros and continuations, which can of course be sent in messages like any other first class object, enabling such operations as task migration and dynamic code update.

An important objective was that it should be flexible enough to allow the programmer to easily build and experiment with libraries providing higher-level distribution primitives and frameworks, so that we can share and reuse more of the design and implementation between applications. Another important objective was that the basic concurrency model should have sufficiently clean semantic properties to make it possible to write simple yet robust code on top of it. Only by attaining those two objectives can we hope to build higher layers of abstractions that are themselves clean, maintainable, and reliable.

Sections 2 and 3 present the core concepts of the Termite model, and the various aspects that are a consequence of that model. Section 4 describes the language, followed by extended examples in Sec. 5. Finally, Section 6 presents the current implementation with some performance measurements.

2. Termite's Programming Model

The foremost design philosophy of the Scheme [14] language is the definition of a small, coherent core which is as general and powerful as possible. This justifies the presence of first class closures and continuations in the language: these features are able to abstract data and control, respectively. In designing Termite, we extended this philosophy to concurrency and distribution features. The model must be simple and extensible, allowing the programmer to build his own concurrency abstractions.

Distributed computations consist of multiple concurrent programs running in usually physically separate spaces and involving data transfer through a potentially unreliable network. In order to model this reality, the concurrency model used in Termite views the computation as a set of isolated sequential processes which are uniquely identifiable across the distributed system. They communicate with each other by exchanging messages. Failure is reflected in Termite by the uncertainty associated with the transmission of

a message: there is no guarantee that a message sent will ever be delivered.

The core features of Termite's model are: isolated sequential processes, message passing, and failure.

2.1 Isolated sequential processes

Termite processes are lightweight. There could be hundreds of thousands of them in a running system. Since they are an important abstraction in the language, the programmer should not consider their creation as costly. She should use them freely to model the problems at hand.

A Termite process executes in the context of a *node*. Nodes are identified with a *node identifier* that contains information to locate a node physically and connect to it (see Sec. 3.4 for details). The procedure `spawn` creates and starts a new process on the node of the parent process.

Termite processes are identified with *process identifiers* or *pids*. *Pids* are *universally unique*. We make the distinction here between *globally unique*, which means unique at the node level, and *universally unique*, which means unique at the whole distributed network level. A *pid* is therefore a reference to a process and contains enough information to determine the node on which the process is located. It is important to note that there is no guarantee that a *pid* refers to a process that is reachable or still alive.

Termite enforces strong isolation between each of the processes: it is impossible for a process to directly access the memory space of another process. This is meant to model the reality of a physically distributed system, and has the advantage of avoiding the problems relative to sharing memory space between processes. This also avoids having to care about mutual exclusion at the language level. There is no need for mutexes and condition variables. Another consequence of that model is that there is no need for a distributed garbage collector, since there cannot be any foreign reference between two nodes's memory spaces. On the other hand, a live process might become unreachable, causing a resource leak: this part of the resource management needs to be done manually.

2.2 Sending and receiving messages

Processes interact by exchanging messages. Each process has a single mailbox in which Termite stores messages in the order in which it receives them. This helps keep the model simple since it saves us from introducing concepts like mailboxes or ports.

In Termite, a message can be any serializable first class value. It can be an atomic value such as a number or a symbol, or a compound value such as a list, record, or continuation, as long as it contains only serializable values.

The message sending operation is asynchronous. When a process sends a message, this is done without the process blocking.

The message retrieval operation is synchronous. A process attempting to retrieve a message from its mailbox will block if no message is available.

Here is an example showing the basic operations used in Termite: A process *A* spawns a new process *B*; The process *B* sends a message to *A*; The process *A* waits until it receives it.

```
(let ((me (self)))
  (spawn
    (lambda ()
      (! me "Hello, world!"))))

(?)                                  ⟹ "Hello, world!"
```

The procedure `self` returns the *pid* of the current process. The procedure `!` is the *send message* operation, while the procedure `?` is the *retrieve the next mailbox message* operation.

2.3 Failure

The unreliability of the physical, "real world" aspects of a distributed computation makes it necessary for that computation to pay close attention to the possibility of failure. A computation run on a single computer with no exterior communication generally does not have to care whether the computer crashes. This is not the case in a distributed setting, where some parts of the computation might go on even in the presence of hardware failure or if the network connection goes down. In order to model failure, sending a message in Termite is an unreliable operation. More specifically, the semantics of the language do not specify how much time a message will take to reach its destination and it may even never reach it, e.g. because of some hardware failure or excessive load somewhere along the way. Joe Armstrong has called this *send and pray semantics* [2].

Since the transmission of a message is unreliable, it is generally necessary for the application to use a protocol with *acknowledgments* to check that the destination has received the message . The burden of implementing such a protocol is left to the application because there are several ways to do it, each with an impact on the way the application is organized. If no acknowledgment is received within a certain time frame, then the application will take some action to recover from the failure. In Termite the mechanism for handling the waiting period is to have an optional timeout for the amount of time to wait for messages. This is a basic mechanism on which we can build higher level failure handling.

3. Peripheral Aspects

Some other Termite features are also notable. While they are not core features, they come naturally when considering the basic model. The most interesting of those derived features are serialization, how to deal with mutation, exception handling and the naming of computers and establishing network connections to them.

3.1 Serialization

There should be no restrictions on the type of data that can constitute a message. Therefore, it is important that the runtime system of the language supports serialization of every first class value in the language, including closures and continuations.

But this is not always possible. Some first class values in Scheme are hard to serialize meaningfully, like ports and references to physical devices. It will not be possible to serialize a closure or a continuation if it has a direct reference to one of these objects in their environment.

To avoid having references to non-serializable objects in the environment, we build *proxies* to those objects by using processes, so that the serialization of such an object will be just a pid. Therefore, Termite uses processes to represent ports (like open files) or references to physical devices (like the mouse and keyboard).

Abstracting non-serializable objects as processes has two other benefits. First, it enables the creation of interesting abstractions. For example, a click of the mouse will send a message to some "mouse listener", sending a message to the process proxying the standard output will print it, etc. Secondly, this allows us to access non-movable resources transparently through the network.

3.2 Explicit mutation

To keep the semantics clean and simplify the implementation, mutation of variables and data structures is not available. This allows the implementation of message-passing within a given computer without having to copy the content of the message.

For this reason, Termite forbids explicit mutation in the system (as with the special form `set!` and procedures `set-car!`, `vector-set!`, etc.) This is not as big a limitation as it seems at

first. It is still possible to replace or simulate mutation using processes. We just need to abstract state using messages and suspended processes. This is a reasonable approach because processes are lightweight. An example of a mutable data structure implemented using a process is given in Section 4.6.

3.3 Exception handling

A Termite exception can be any first class value. It can be *raised* by an explicit operation, or it can be the result of a software error (like division by zero or a type error).

Exceptions are dealt with by installing dynamically scoped handlers. Any exception raised during execution will invoke the handler with the exception as a parameter. The handler can either choose to manage that exceptional condition and resume execution or to raise it again. If it raises the exception again, it will invoke the nearest encapsulating handler. Otherwise, the point at which execution resumes depends on the handler: an *exception-handler* will resume execution at the point the exception was raised, whereas an *exception-catcher* will resume execution at the point that the handler was installed.

If an exception propagates to the outer scope of the process (i.e. an uncaught exception), the process dies. In order to know who to notify of such a circumstance, each process has what we call *links* to other processes. When a process dies and it is *linked* to other processes, Termite propagates the exception to those processes. Links between processes are directed. A process which has an outbound link to another process will send any uncaught exception to the other process. Note that exception propagation, like all communication, is unreliable. The implementation will make an extra effort when delivering an exception since that kind of message may be more important for the correct execution of the application.

Receiving an exception causes it to be raised in the receiving process at the moment of the next *message retrieve* operation by that process.

Links can be established in both directions between two processes. In that situation the link is said to be *bidirectional*. The direction of the link should reflect the relation between the two processes. In a supervisor-worker relation, we will use a bidirectional link since both the supervisor and the worker need to learn about the death of the other (the supervisor so it may restart the worker, the worker so it can stop executing). In a monitor-worker relation where the monitor is an exterior observer to the worker, we will use an outbound link from the worker since the death of the monitor should not affect the worker.

3.4 Connecting nodes

Termite processes execute on nodes. Nodes connect to each other when needed in order to exchange messages. The current practice in Termite is to uniquely identify nodes by binding them to an IP address and a TCP port number. Node references contain exactly that information and therefore it is possible to reach a node from the information contained in the reference. Those references are built using the `make-node` procedure.

Termite's distributed system model is said to be *open*: nodes can be added or removed from a distributed computation at any time. Just like it is possible to spawn a process on the current node, it is possible to spawn a process on a remote node by using the `remote-spawn` procedure. This is one of the key features that enable distribution.

The concept of global environment as it exists in Scheme is tied to a node. A variable referring to the global environment will resolve to the value tied to that variable on the node on which the process is currently executing.

3.5 Tags

A process may make multiple concurrent requests to another process. Also, replies to requests may come out of order (and even from a completely different process, e.g. if the request was forwarded). In those cases, it can be difficult to sort out which reply corresponds to which request. For this purpose, Termite has a universally unique reference data type called *tag*. When needed, the programmer can then uniquely mark each new request with a new *tag*, and copy the tag into the replies, to unequivocally indicate which reply corresponds to which request. Note that this can be necessary even when there is apparently only one request pending, since the process may receive a spurious delayed reply to some earlier request which had timed out.

4. The Termite Language

This section introduces the Termite language through examples. For the sake of simplicity those examples assume that messages will always be delivered (no failure) and always in the same order that they were sent.

The fundamental operations of Termite are:

`(spawn` *`fun`*`)`: create a process running *fun* and return its *pid*.

`(!` *`pid msg`*`)`: send message *msg* to process *pid*.

`(?` `[`*`timeout`* `[`*`default`*`]]``)`: fetch a message from the mailbox.

4.1 Making a "server" process

In the following code, we create a process called `pong-server`. This process will reply with the symbol `pong` to any message that is a list of the form `(`*`pid`* `ping)` where *pid* refers to the originating process. The Termite procedure `self` returns the *pid* of the current process.

```
(define pong-server
  (spawn
    (lambda ()
      (let loop ()
        (let ((msg (?)))
          (if (and (list? msg)
                   (= (length msg) 2)
                   (pid? (car msg))
                   (eq? (cadr msg) 'ping))
              (let ((from (car msg)))
                (! from 'pong)
                (loop))
              (loop)))))))

(! pong-server (list (self) 'ping))

(?)                                ⟹ pong
```

4.2 Selective message retrieval

While the `?` procedure retrieves the next available message in the process' mailbox, sometimes it can be useful to be able to choose the message to retrieve based on a certain criteria. The selective message retrieval procedure is `(??` *`pred`* `[`*`timeout`* `[`*`default`*`]]`). It retrieves the first message in the mailbox which satisfies the predicate *pred*. If none of the messages in the mailbox satisfy *pred*, then it waits until one arrives that does or until the timeout is hit.

Here is an example of the `??` procedure in use:

```
(! (self) 1)
(! (self) 2)
```

```
(! (self) 3)

(?)                                ⟹ 1
(?? odd?)                          ⟹ 3
(?)                                ⟹ 2
```

4.3 Pattern matching

The previous `pong-server` example showed that ensuring that a message is well-formed and extracting relevant information from it can be quite tedious. Since those are frequent operations, Termite offers an ML-style pattern matching facility.

Pattern matching is implemented as a special form called `recv`, conceptually built on top of the `??` procedure. It has two simultaneous roles: selective message retrieval and data destructuring. The following code implements the same functionality as the previous *pong server* but using `recv`:

```
(define better-pong-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from 'ping)        ; pattern to match
           (where (pid? from)) ; constraint
           (! from 'pong)))    ; action
        (loop)))))
```

The use of `recv` here only has one clause, with the pattern `(from 'ping)` and an additional side condition (also called *where clause*) `(pid? from)`. The pattern constrains the message to be a list of two elements where the first can be anything (ignoring for now the subsequent side condition) and will be bound to the variable `from`, while the second has to be the symbol `ping`. There can of course be several clauses, in which case the first message that matches one of the clauses will be processed.

4.4 Using timeouts

Timeouts are the fundamental way to deal with unreliable message delivery. The operations for receiving messages (ie. `?`, `??`) can optionally specify the maximum amount of time to wait for the reception of a message as well as a default value to return if this timeout is reached. If no timeout is specified, the operation will wait forever. If no default value is specified, the `timeout` symbol will be raised as an exception. The `recv` special form can also specify such a timeout, with an `after` clause which will be selected after no message matched any of the other clauses for the given amount of time.

```
(! some-server (list (self) 'request argument))

(? 10) ; waits for a maximum of 10 seconds
;; or, equivalently:
(recv
 (x x)
 (after 10 (raise 'timeout)))
```

4.5 Remote procedure call

The procedure `spawn` takes a thunk as parameter, creates a process which evaluates this thunk, and returns the *pid* of this newly created process. Here is an example of an RPC server to which uniquely identified requests are sent. In this case a synchronous call to the server is used:

```
(define rpc-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag ('add a b))
           (! from (list tag (+ a b)))))
        (loop)))))

(let ((tag (make-tag)))
  (! rpc-server (list (self)
                      tag
                      (list 'add 21 21)))
  (recv
    ;; note the reference to tag in
    ;; the current lexical scope
    ((,tag reply) reply)))      ⟹ 42
```

The pattern of implementing a synchronous call by creating a tag and then waiting for the corresponding reply by testing for tag equality is frequent. This pattern is abstracted by the procedure `!?`. The following call is equivalent to the last `let` expression in the previous code:

```
(!? rpc-server (list 'add 21 21))
```

Note that the procedure `!?` can take optional *timeout* and *default* arguments like the message retrieving procedures.

4.6 Mutable data structure

While Termite's native data structures are immutable, it is still possible to implement mutable data structures using processes to represent state. Here is an example of the implementation of a mutable cell:

```
(define (make-cell content)
  (spawn
    (lambda ()
      (let loop ((content content))
        (recv
          ((from tag 'ref)
           (! from (list tag content))
           (loop content))

          (('set! content)
           (loop content)))))))

(define (cell-ref cell)
  (!? cell 'ref))

(define (cell-set! cell value)
  (! cell (list 'set! value)))
```

4.7 Dealing with exceptional conditions

Explicitly signaling an exceptional condition (such as an error) is done using the `raise` procedure. Exception handling is done using one of the two procedures `with-exception-catcher` and `with-exception-handler`, which install a dynamically scoped exception handler (the first argument) for the duration of the evaluation of the body (the other arguments).

After invoking the handler on an exception, the procedure `with-exception-catcher` will resume execution at the point where the handler was installed. `with-exception-handler`, the alternative procedure, will resume execution at the point where the exception was raised. The following example illustrates this difference:

```
(list
 (with-exception-catcher
  (lambda (exception) exception)
  (lambda ()
    (raise 42) ; this will not return
    123))                          ⟹ (42)

(list
 (with-exception-handler
  (lambda (exception) exception)
  (lambda ()
    (raise 42) ; control will resume here
    123))                          ⟹ (123)
```

The procedure `spawn-link` creates a new process, just like `spawn`, but this new process is bidirectionally linked with the current process. The following example shows how an exception can propagate through a link between two processes:

```
(catch
  (lambda (exception) #t)
  (spawn (lambda () (raise 'error)))
  (? 1 'ok)
  #f)                              ⟹ #f

(catch
  (lambda (exception) #t)
  (spawn-link (lambda () (raise 'error)))
  (? 1 'ok)
  #f)                              ⟹ #t
```

4.8 Remotely spawning a process

The function to create a process on another node is `remote-spawn`. Here is an example of its use:

```
(define node (make-node "example.com" 3000))

(let ((me (self)))
  (remote-spawn node
    (lambda ()
      (! me 'boo))))               ⟹ a-pid

(?)                                ⟹ boo
```

Note that it is also possible to establish links to remote processes. The `remote-spawn-link` procedure atomically spawns and links the remote process:

```
(define node (make-node "example.com" 3000))

(catch
  (lambda (exception) exception)
  (let ((me (self)))
    (remote-spawn-link node
      (lambda ()
        (raise 'error))))
  (? 2 'ok))                       ⟹ error
```

4.9 Abstractions built using continuations

Interesting abstractions can be defined using `call/cc`. In this section we give as an example process migration, process cloning, and dynamic code update.

Process migration is the act of moving a computation from one node to another. The presence of serializable continuations in Termite makes it easy. Of the various possible forms of process migration, two are shown here. The simplest form of migration, called here `migrate-task`, is to move a process to another node, abandoning messages in its mailbox and current links behind. For that we capture the continuation of the current process, start a new process on a remote node which invokes this continuation, and then terminate the current process:

```
(define (migrate-task node)
  (call/cc
   (lambda (k)
     (remote-spawn node (lambda () (k #t)))
     (halt!))))
```

A different kind of migration (`migrate/proxy`), which might be more appropriate in some situations, will take care to leave a process behind (a *proxy*) which will forward messages sent to it to the new location. In this case, instead of stopping the original process we make it execute an endless loop which forwards to the new process every message received:

```
(define (migrate/proxy node)
  (define (proxy pid)
    (let loop ()
      (! pid (?))
      (loop)))
  (call/cc
   (lambda (k)
     (proxy
      (remote-spawn-link
       node
       (lambda () (k #t)))))))
```

Process cloning is simply creating a new process from an existing process with the same state and the same behavior. Here is an example of a process which will reply to a `clone` message with a thunk that makes any process become a "clone" of that process:

```
(define original
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag 'clone)
           (call/cc
            (lambda (clone)
              (! from (list tag (lambda ()
                                  (clone #t))))))))
        (loop)))))

(define clone (spawn (!? original 'clone)))
```

Updating code dynamically in a running system can be very desirable, especially with long-running computations or in high-availability environments. Here is an example of such a dynamic code update:

```
(define server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          (('update k)
           (k #t))

          ((from tag 'ping)
           (! from (list tag 'gnop))))  ; bug
        (loop)))))
```

```
(define new-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((’update k)
           (k #t))

          ((from tag ’clone)
           (call/cc
            (lambda (k)
              (! from (list tag k)))))

          ((from tag ’ping)
           (! from (list tag ’pong))))  ; fixed
        (loop)))))

(!? server ’ping)                ⟹ gnop

(! server (list ’update (!? new-server ’clone)))

(!? server ’ping)                ⟹ pong
```

Note that this allows us to build a new version of a running process, test and debug it separately and when it is ready replace the running process with the new one. Of course this necessitates cooperation from the process whose code we want to replace (it must understand the `update` message).

5. Examples

One of the goals of Termite is to be a good framework to experiment with abstractions of patterns of concurrency and distributed protocols. In this section we present three examples: first a simple load-balancing facility, then a technique to abstract concurrency in the design of a server and finally a way to transparently “robustify” a process.

5.1 Load Balancing

This first example is a simple implementation of a load-balancing facility. It is built from two components: the first is a *meter supervisor*. It is a process which supervises workers (called *meters* in this case) on each node of a cluster in order to collect load information. The second component is the work dispatcher: it receives a closure to evaluate, then dispatches that closure for evaluation to the node with the lowest current load.

Meters are very simple processes. They do nothing but send the load of the current node to their supervisor every second:

```
(define (start-meter supervisor)
  (let loop ()
    (! supervisor
       (list ’load-report
             (self)
             (local-loadavg)))
    (recv (after 1 ’ok)) ; pause for a second
    (loop)))
```

The supervisor creates a dictionary to store current load information for each meter it knows about. It listens for the update messages and replies to requests for the node in the cluster with the lowest current load and to requests for the average load of all the nodes. Here is a simplified version of the supervisor:

```
(define (meter-supervisor meter-list)
  (let loop ((meters (make-dict)))
    (recv
      ((’load-report from load)
       (loop (dict-set meters from load)))
      ((from tag ’minimum-load)
       (let ((min (find-min (dict->list meters))))
         (! from (list tag (pid-node (car min)))))
       (loop dict))
      ((from tag ’average-load)
       (! from (list tag
                     (list-average
                      (map cdr
                           (dict->list meters)))))
       (loop dict)))))

(define (minimum-load supervisor)
  (!? supervisor ’minimum-load))

(define (average-load supervisor)
  (!? supervisor ’average-load))
```

And here is how we may start such a supervisor:

```
(define (start-meter-supervisor)
  (spawn
    (lambda ()
      (let ((supervisor (self)))
        (meter-supervisor
         (map
          (lambda (node)
            (spawn
              (migrate node)
              (start-meter supervisor)))
          *node-list*))))))
```

Now that we can establish what is the current load on nodes in a cluster, we can implement load balancing. The *work dispatching server* receives a thunk, and migrates its execution to the currently least loaded node of the cluster. Here is such a server:

```
(define (start-work-dispatcher load-server)
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag (’dispatch thunk))
           (let ((min-loaded-node
                  (minimum-load load-server)))
             (spawn
               (lambda ()
                 (migrate min-loaded-node)
                 (! from (list tag (thunk))))))))
        (loop)))))

(define (dispatch dispatcher thunk)
  (!? dispatcher (list ’dispatch thunk)))
```

It is then possible to use the procedure `dispatch` to request execution of a thunk on the most lightly loaded node in a cluster.

5.2 Abstracting Concurrency

Since building distributed applications is a complex task, it is particularly beneficial to abstract common patterns of concurrency. An example of such a pattern is a server process in a client-server organization. We use Erlang’s concept of behaviors to do that: behaviors are implementations of particular patterns of concurrent interaction.

The behavior given as example in this section is derived from the *generic server* behavior. A generic server is a process that can be started, stopped and restarted, and answers RPC-like requests.

The behavior contains all the code that is necessary to handle the message sending and retrieving necessary in the implementation of a server. The behavior is only the generic framework. To create a server we need to parameterize the behavior using a *plugin* that describes the server we want to create. A plugin contains closures (often called *callbacks*) that the generic code calls when certain events occur in the server.

A plugin only contains sequential code. All the code having to deal with concurrency and passing messages is in the generic server's code. When invoking a callback, the current server state is given as an argument. The reply of the callback contains the potentially modified server code.

A generic server plugin contains four closures. The first is for server initialization, called when creating the server. The second is for procedure calls to the server: the closure dispatches on the term received in order to execute the function call. Procedure calls to the server are synchronous. The third closure is for *casts*, which are asynchronous messages sent to the server in order to do management tasks (like restarting or stopping the server). The fourth and last closure is called when terminating the server.

Here is an example of a generic server plugin implementing a key/value server:

```
(define key/value-generic-server-plugin
  (make-generic-server-plugin
   (lambda ()                           ; INIT
     (print "Key-Value server starting")
     (make-dict))

   (lambda (term from state)            ; CALL
     (match term
       ((’store key val)
        (dict-set! state key val)
        (list ’reply ’ok state))

       ((’lookup key)
        (list ’reply (dict-ref state key) state))))

   (lambda (term state)                 ; CAST
     (match term
       (’stop (list ’stop ’normal state))))

   (lambda (reason state)               ; TERMINATE
     (print "Key-Value server terminating"))))
```

It is then possible to access the functionality of the server by using the generic server interface:

```
(define (kv:start)
  (generic-server-start-link
   key/value-generic-server-plugin))

(define (kv:stop server)
  (generic-server-cast server ’stop))

(define (kv:store server key val)
  (generic-server-call server (list ’store key val)))

(define (kv:lookup server key)
  (generic-server-call server (list ’lookup key)))
```

Using such concurrency abstractions helps in building reliable software, because the software development process is less error-prone. We reduce complexity at the cost of flexibility.

5.3 Fault Tolerance

Promoting the writing of simple code is only a first step in order to allow the development of robust applications. We also need to be able to handle system failures and software errors. Supervisors are another kind of behavior in the Erlang language, but we use a slightly different implementation from Erlang's. A *supervisor* process is responsible for supervising the correct execution of a *worker* process. If there is a failure in the worker, the supervisor restarts it if necessary.

Here is an example of use of such a supervisor:

```
(define (start-pong-server)
  (let loop ()
    (recv
      ((from tag ’crash)
       (! from (list tag (/ 1 0))))
      ((from tag ’ping)
       (! from (list tag ’pong))))
    (loop)))

(define robust-pong-server
  (spawn-thunk-supervised start-pong-server))

(define (ping server)
  (!? server ’ping 1 ’timeout))

(define (crash server)
  (!? server ’crash 1 ’crashed))

(define (kill server)
  (! server ’shutdown))

(print (ping robust-pong-server))
(print (crash robust-pong-server))
(print (ping robust-pong-server))
(kill robust-pong-server)
```

This generates the following trace (note that the messages prefixed with `info:` are debugging messages from the supervisor) :

```
(info: starting up supervised process)
pong
(info: process failed)
(info: restarting...)
(info: starting up supervised process)
crashed
pong
(info: had to terminate the process)
(info: halting supervisor)
```

The call to `spawn-thunk-supervised` return the *pid* of the supervisor, but any message sent to the supervisor is sent to the worker. The supervisor is then mostly transparent: interacting processes do not necessarily know that it is there.

There is one special message which the supervisors intercepts, and that consists of the single symbol `shutdown`. Sending that message to the supervisor makes it invoke a `shutdown` procedure that requests the process to end its execution, or terminate it if it does not collaborate. In the previous trace, the "had to terminate the process" message indicates that the process did not acknowledge the request to end its execution and was forcefully terminated.

A supervisor can be parameterized to set the acceptable restart frequency tolerable for a process. A process failing more often than a certain limit is shut down. It is also possible to specify the delay that the supervisor will wait for when sending a shutdown request to the worker.

The abstraction shown in this section is useful to construct a fault-tolerant server. A more general abstraction would be able to supervise multiple processes at the same time, with a policy determining the relation between those supervised processes (should the supervisor restart them all when a single process fails or just the failed process, etc.).

5.4 Other Applications

As part of Termite's development, we implemented two non-trivial distributed applications with Termite. *Dynamite* is a framework for developing dynamic AJAX-like web user-interfaces. We used Termite processes to implement the web-server side logic, and we can manipulate user-interface components directly from the server-side (for example through the *repl*). *Schack* is an interactive multiplayer game using Dynamite for its GUI. Players and monsters move around in a virtual world, they can pick up objects, use them, etc. The rooms of the world, the players and monsters are all implemented using Termite processes which interact.

6. The Termite Implementation

The Termite system was implemented on top of the Gambit-C Scheme system [6]. Two features of Gambit-C were particularly helpful for implementing the system: lightweight threads and object serialization.

Gambit-C supports lightweight prioritized threads as specified by SRFI 18 [7] and SRFI 21 [8]. Each thread descriptor contains the thread's continuation in the same linked frame representation used by first class continuations produced by `call/cc`. Threads are suspended by capturing the current continuation and storing it in the thread descriptor. The space usage for a thread is thus dependent on the depth of its continuation and the objects it references at that particular point in the computation. The space efficiency compares well with the traditional implementation of threads which preallocates a block of memory to store the stack, especially in the context of a large number of small threads. On a 32 bit machine the total heap space occupied by a trivial suspended thread is roughly 650 bytes. A single shared heap is used by all the threads for all allocations including continuations (see [9] for details). Because the thread scheduler uses scalable data structures (red-black trees) to represent priority queues of runnable, suspended and sleeping threads, and threads take little space, it is possible to manage millions of threads on ordinary hardware. This contributes to make the Termite model practically insensitive to the number of threads involved.

Gambit-C supports serialization for an interesting subset of objects including closures and continuations but not ports, threads and foreign data. The serialization format preserves sharing, so even data with cycles can be serialized. We can freely mix interpreted code and compiled code in a given program. The Scheme interpreter, which is written in Scheme, is in fact compiled code in the Gambit-C runtime system. Interpreted code is represented with common Scheme objects (vectors, closures created by compiled code, symbols, etc.). Closures use a flat representation, i.e. a closure is a specially tagged vector containing the free variables and a pointer to the entry point in the compiled code. Continuation frames use a similar representation, i.e. a specially tagged vector containing the continuation's free variables, which include a reference to the parent continuation frame, and a pointer to the return point in the compiled code. When Scheme code is compiled with Gambit-C's *block* option, which signals that procedures defined at top-level are never redefined, entry points and return points are identified using the name of the procedure that contains them and the integer index of the control point within that procedure. Serialization of closures and continuations created by compiled code is thus possible as long as they do not refer to non-serializable objects and the block option is used. However, the Scheme program performing the deserialization must have the same compiled code, either statically linked or dynamically loaded. Because the Scheme interpreter in the Gambit-C runtime is compiled with the block option, we can always serialize closures and continuations created by interpreted code and we can deserialize them in programs using the same version of Gambit-C. The serialization format is machine independent (endianness, machine word size, instruction set, memory layout, etc.) and can thus be deserialized on any machine. Continuation serialization allows the implementation of process migration with `call/cc`.

For the first prototype of Termite we used the Gambit-C system as-is. During the development process various performance problems were identified. This prompted some changes to Gambit-C which are now integrated in the official release:

- **Mailboxes**: Each Gambit-C thread has a mailbox. Predefined procedures are available to probe the messages in the mailbox and extract messages from the mailbox. The operation to advance the probe to the next message optionally takes a timeout. This is useful for implementing Termite's time limited receive operations.
- **Thread subtyping**: There is a `define-type-of-thread` special form to define subtypes of the builtin thread type. This is useful to attach thread local information to the thread, in particular the process links.
- **Serialization**: Originally serialization used a textual format compatible with the standard datum syntax but extended to all types and with the SRFI 38 [5] notation for representing cycles. We added hash tables to greatly improve the speed of the algorithm for detecting shared data. We improved the compactness of the serialized objects by using a binary format. Finally, we parameterized the serialization and deserialization procedures (`object->u8vector` and `u8vector->object`) with an optional conversion procedure applied to each subobject visited during serialization or constructed during deserialization. This allows the program to define serialization and deserialization methods for objects such as ports and threads which would otherwise not be serializable.
- **Integration into the Gambit-C runtime**: To correctly implement tail-calls in C, Gambit-C uses computed gotos for intra-module calls but trampolines to jump from one compilation unit to another. Because the Gambit-C runtime and the user program are distributed over several modules, there is a relatively high cost for calling procedures in the runtime system from the user program. When the Termite runtime system is in a module of its own, calls to some Termite procedures must cross two module boundaries (user program to Termite runtime, and Termite runtime to Gambit-C runtime). For this reason, integrating the Termite runtime in the thread module of the Gambit-C runtime enhances execution speed (this is done simply by adding `(include "termite.scm")` at the end of the thread module).

7. Experimental Results

In order to evaluate the performance of Termite, we ran some benchmark programs using Termite version 0.9. When possible, we compared the two systems by executing the equivalent Erlang program using Erlang/OTP version R11B-0, compiled with SMP support disabled. Moreover, we also rewrote some of the benchmarks directly in Gambit-C Scheme and executed them with version 4.0 beta 18 to evaluate the overhead introduced by Termite. In all cases we compiled the code, and no optimization flags were given to the compilers. We used the compiler GCC version 4.0.2 to compile Gambit-C, and we specified the configuration option "–

enable-single-host" for the compilation. We ran all the benchmarks on a GNU/Linux machine with a 1 GHz AMD Athlon 64, 2GB RAM and a 100Mb/s Ethernet, running kernel version 2.6.10.

7.1 Basic benchmarks

Simple benchmarks were run to compare the general performance of the systems on code which does not require concurrency and distribution. The benchmarks evaluate basic features like the cost of function calls and memory allocation.

The following benchmarks were used:

- The recursive Fibonacci and Takeuchi functions, to estimate the cost of function calls and integer arithmetic,
- Naive list reversal, to strain memory allocation and garbage collection,
- Sorting a list of random integers using the *quicksort* algorithm,
- String matching using the Smith Waterman algorithm.

The results of those benchmarks are given in Figure 1. They show that Termite is generally 2 to 3.5 times faster than Erlang/OTP. The only exception is for *nrev* which is half the speed of Erlang/OTP due to the overhead of Gambit-C's interrupt polling approach.

Test	Erlang (s)	Termite (s)
fib (34)	1.83	0.50
tak (27, 18, 9)	1.00	0.46
nrev (5000)	0.29	0.53
qsort (250000)	1.40	0.56
smith (600)	0.46	0.16

Figure 1. Basic benchmarks.

7.2 Benchmarks for concurrency primitives

We wrote some benchmarks to evaluate the relative performance of Gambit-C, Termite, and Erlang for primitive concurrency operations, that is process creation and exchange of messages.

The first two benchmarks stress a single feature. The first (*spawn*) creates a large number of processes. The first process creates the second and terminates, the second creates the third and terminates, and so on. The last process created terminates the program. The time for creating a single process is reported. In the second benchmark (*send*), a process repeatedly sends a message to itself and retrieves it. The time needed for a single message send and retrieval is reported. The results are given in Figure 2. Note that neither program causes any process to block. We see that Gambit-C and Termite are roughly twice the speed of Erlang/OTP for process creation, and roughly 3 times slower than Erlang/OTP for message passing. Termite is somewhat slower than Gambit-C because of the overhead of calling the Gambit-C concurrency primitives from the Termite concurrency primitives, and because Termite processes contain extra information (list of linked processes).

Test	Erlang (μs)	Gambit (μs)	Termite (μs)
spawn	1.57	0.63	0.91
send	0.08	0.22	0.27

Figure 2. Benchmarks for concurrency primitives.

The third benchmark (*ring*) creates a ring of 250 thousand processes on a single node. Each process receives an integer and then sends this integer minus one to the next process in the ring. When the number received is 0, the process terminates its execution after sending 0 to the next process. This program is run twice with a different initial number (K). Each process will block a total of $\lceil K/250000 \rceil + 1$ times (once for $K = 0$ and 5 times for $K = 1000000$).

With $K = 0$ it is mainly the ring creation and destruction time which is measured. With $K = 1000000$, message passing and process suspension take on increased importance. The results of this benchmark are given in Figure 3. Performance is given in microseconds per process. A lower number means better performance.

K	Erlang (μs)	Gambit (μs)	Termite (μs)
0	6.64	4.56	7.84
1000000	7.32	14.36	15.48

Figure 3. Performance for ring of 250000 processes

We can see that all three systems have similar performance for process creation; Gambit-C is slightly faster than Erlang and Termite is slightly slower. The performance penalty for Termite relatively to Gambit-C is due in part to the extra information Termite processes must maintain (like a list of links) and the extra test on message sends to determine whether they are intended for a local or a remote process. Erlang shows the best performance when there is more communication between processes and process suspension.

7.3 Benchmarks for distributed applications

7.3.1 "Ping-Pong" exchanges

This benchmark measures the time necessary to send a message between two processes exchanging *ping-pong* messages. The program is run in three different situations: when the two processes are running on the same node, when the processes are running on different nodes located on the same computer and when the processes are running on different nodes located on two computers communicating across a local area network. In each situation, we vary the volume of the messages sent between the processes by using lists of small integers of various lengths. The measure of performance is the time necessary to send and receive a single message. The lower the value, the better the performance.

List length	Erlang (μs)	Gambit (μs)	Termite (μs)
0	0.20	0.67	0.75
10	0.31	0.67	0.75
20	0.42	0.67	0.74
50	0.73	0.68	0.75
100	1.15	0.66	0.74
200	1.91	0.67	0.75
500	4.40	0.67	0.75
1000	8.73	0.67	0.75

Figure 4. Local ping-pong: Measure of time necessary to send and receive a message of variable length between two processes running on the same node.

The *local ping-pong* benchmark results in Figure 4 illustrate an interesting point: when the volume of messages grows, the performance of the Erlang system diminishes, while the performance of Termite stays practically the same. This is due to the fact that the Erlang runtime uses a separate heap per process, while the Gambit-C runtime uses a shared heap approach.

List length	Erlang (μs)	Termite (μs)
0	53	145
10	52	153
20	52	167
50	54	203
100	55	286
200	62	403
500	104	993
1000	177	2364

Figure 5. Inter-node ping-pong: Measure of time necessary to send and receive a message of variable length between two processes running on two different nodes on the same computer.

The *inter-node ping-pong* benchmark exercises particularly the serialization code, and the results in Figure 5 show clearly that Erlang's serialization is significantly more efficient than Termite's. This is expected since serialization is a relatively new feature in Gambit-C that has not yet been optimized. Future work should improve this aspect.

List length	Erlang (μs)	Termite (μs)
0	501	317
10	602	337
20	123	364
50	102	437
100	126	612
200	176	939
500	471	1992
1000	698	3623

Figure 6. Remote ping-pong: Measure of time necessary to send and receive a message of variable length between two processes running on two different computers communicating through the network.

Finally, the *remote ping-pong* benchmark additionally exercises the performance of the network communication code. The results are given in Figure 6. The difference with the previous program shows that Erlang's networking code is also more efficient than Termite's by a factor of about 2.5 for large messages. This appears to be due to more optimized networking code as well as a more efficient representation on the wire, which comes back to the relative youth of the serialization code. The measurements with Erlang show an anomalous slowdown for small messages which we have not been able to explain. Our best guess is that Nagle's algorithm gets in the way, whereas Termite does not suffer from it because it explicitly disables it.

7.3.2 Process migration

We only executed this benchmark with Termite, since Erlang does not support the required functionality. This program was run in three different configurations: when the process migrates on the same node, when the process migrates between two nodes running on the same computer, and when the process migrates between two nodes running on two different computers communicating through a network. The results are given in Figure 7. Performance is given in number of microseconds necessary for the migration. A lower value means better performance.

The results show that the main cost of a migration is in the serialization and transmission of the continuation. Comparatively, capturing a continuation and spawning a new process to invoke it is almost free.

Migration	Termite (μs)
Within a node	4
Between two local nodes	560
Between two remote nodes	1000

Figure 7. Time required to migrate a process.

8. Related Work

The **Actors** model is a general model of concurrency that has been developed by Hewitt, Baker and Agha [13, 12, 1]. It specifies a concurrency model where independent actors concurrently execute code and exchange messages. Message delivery is guaranteed in the model. Termite might be considered as an "impure" actor language, because it does not adhere to the strict "everything is an actor" model since only processes are actors. It also diverges from that model by the unreliability of the message transmission operation.

Erlang [3, 2] is a distributed programming system that has had a significant influence on this work. Erlang was developed in the context of building telephony applications, which are inherently concurrent. The idea of multiple lightweight isolated processes with unreliable asynchronous message transmission and controlled error propagation has been demonstrated in the context of Erlang to be useful and efficient. Erlang is a dynamically-typed semi-functional language similar to Scheme in many regards. Those characteristics have motivated the idea of integrating Erlang's concurrency ideas to a Lisp-like language. Termite notably adds to Erlang first class continuations and macros. It also features directed links between processes, while Erlang's links are always bidirectionals.

Kali [4] is a distributed implementation of Scheme. It allows the migration of higher-order objects between computers in a distributed setting. It uses a shared-memory model and requires a distributed garbage collector. It works using a centralized model where a node is supervising the others, while Termite has a peer-to-peer model. Kali does not feature a way to deal with network failure, while that is a fundamental aspect of Termite. It implements efficient communication by keeping a cache of objects and lazily transmitting closure code, which are techniques a Termite implementation might benefit from.

The Tube [11] demonstrates a technique to build a distributed programming system on top of an existing Scheme implementation. The goal is to have a way to build a distributed programming environment without changing the underlying system. It relies on the "code as data" property of Scheme and on a custom interpreter able to save state to code represented as S-expressions in order to implement code migration. It is intended to be a minimal addition to Scheme that enables distributed programming. Unlike Termite, it neither features lightweight isolated process nor considers the problems associated with failures.

Dreme [10] is a distributed programming system intended for open distributed systems. Objects are mobile in the network. It uses a shared memory model and implements a fault-tolerant distributed garbage collector. It differs from Termite in that it sends objects to remote processes by reference unless they are explicitly migrated. Those references are resolved transparently across the network, but the cost of operations can be hidden, while in Termite costly operations are explicit. The system also features a User Interface toolkit that helps the programmer to visualize distributed computation.

9. Conclusion

Termite has shown to be an appropriate and interesting language and system to implement distributed applications. Its core model is simple yet allows for the abstraction of patterns of distributed computation.

We built the current implementation on top of the Gambit-C Scheme system. While this has the benefit of giving a lot of freedom and flexibility during the exploration phase, it would be interesting to build from scratch a system with the features described in this paper. Such a system would have to take into consideration the frequent need for serialization, try to have processes as lightweight and efficient as possible, look into optimizations at the level of what needs to be transferred between nodes, etc. Apart from the optimizations it would also benefit from an environment where a more direct user interaction with the system would be possible. We intend to take on those problems in future research while pursuing the ideas laid in this paper.

10. Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Department of Microelectronics and Information Technology, Stockholm, Sweden, December 2003.

[3] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[4] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-Order Distributed Objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, 1995.

[5] Ray Dillinger. SRFI 38: External representation for data with shared structure. `http://srfi.schemers.org/srfi-38/srfi-38.html`.

[6] Marc Feeley. Gambit-C version 4. `http://www.iro.umontreal.ca/~gambit`.

[7] Marc Feeley. SRFI 18: Multithreading support. `http://srfi.schemers.org/srfi-18/srfi-18.html`.

[8] Marc Feeley. SRFI 21: Real-time multithreading support. `http://srfi.schemers.org/srfi-21/srfi-21.html`.

[9] Marc Feeley. A case for the unified heap approach to erlang memory management. *Proceedings of the PLI'01 Erlang Workshop*, September 2001.

[10] Matthew Fuchs. *Dreme: for Life in the Net*. PhD thesis, New York University, Computer Science Department, New York, NY, United States, July 2000.

[11] David A. Halls. *Applying mobile code to distributed systems*. PhD thesis, University of Cambridge, Computer Laboratory, Cambridge, United Kingdom, December 1997.

[12] C. E. Hewitt and H. G. Baker. Actors and continuous functionals. In E. J. Neuhold, editor, *Formal Descriptions of Programming Concepts*. North Holland, Amsterdam, NL, 1978.

[13] Carl E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[14] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.