Termite: a Lisp for Distributed Computing

Guillaume Germain, Marc Feeley, Stefan Monnier
Université de Montréal
Département d'informatique et r.o.
Canada
{germaing, feeley, monnier}@iro.umontreal.ca

ABSTRACT

Termite is a language and system offering a simple and powerful tool for expressing distributed computation. It is based on a message-passing model of concurrency inspired by Erlang, and on a variant of the functional language Scheme.

Our system is an appropriate tool 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 them are proposed. The existence of first-class continuations is exploited in order to allow the expression of high-level concepts such as transparent migration of processes.

We describe the Termite model with its implications and describe sample applications built with Termite. We conclude with a discussion of the current implementation and its performance.

General Terms

Distributed computing in Lisp

Keywords

Distributed computing, Scheme, Lisp, Erlang, Continuations

1. INTRODUCTION

There is a great need for the development of distributed applications. These applications are found under different forms: email, web pages, newsgroups, chat rooms, games, telephony, file swapping, etc. All distributed applications share the property of being constituted from 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 always present in that setting, due to the unreliability of networks and computer hardware.

When building a distributed application, the common practice is to use an ad-hoc approach for each particular problem. Much of the work has to be redone every time: how to serialize the application's data, how to synchronize the computation, how to deal with exceptional conditions like network failures, etc. This incurs an overhead that slows down the development process and introduces numerous bug opportunities.

Instead of having to repeat a part of the work every time, Termite offers a simple yet high-level concurrency model on which reliable distributed applications can be built. 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. An important objective is that it should be flexible enough to allow the programmer to easily build and experiment with custom distributed protocols and applications.

We will first present the core concepts of the Termite model, and the various aspects that are a consequence of that model. The language will then be described, followed by extended examples. Finally, the current implementation will be presented with some performance measurements.

2. THE TERMITE PROGRAMMING MODEL

The foremost design philosophy of the Scheme [12] 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, this philosophy has been extended to concurrency and distribution features. The model must be simple and extensible, allowing the programmer to build his own concurrency abstractions.

Distributed computations are composed of multiple concurrent programs running in physically separated 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, their creation should not be considered costly by the programmer. They should be freely used to model the problems at hand.

A Termite process is executed 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 section 3.4 for details). It is possible to spawn a new process using the special form `spawn`, and to migrate that process to another node.

Termite processes are identified with a *process identifier* or *pid*. *Pid*s 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.

Strong isolation is enforced on 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 situation of a 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' memory spaces. On the other hand, a live process might become unreachable, causing a memory leak.

2.2 Sending and receiving messages

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. Each process has a single mailbox in which messages are stored in the order in which they are delivered.

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 acceptable message is available.

Here is an example showing the basic operators used in Termite. A process A spawns a new process B. The process B sends a message to A. The process A waits until the message is received.

```
(let ((me (self)))
  (spawn
    (! 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 message" operation.

2.3 Failure

Failure is caused by the unreliability of the physical, "real world" aspects of a distributed computation. A computation being executed on a single computer with no exterior communication generally doesn't have to care whether the computer crashes. This isn't 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 [1].

Since the transmission of a message is unreliable, the only way to know for sure if a message got to its destination is to have a protocol where the originating process will wait for an acknowledgment message to be sent back. The mechanism for handling the waiting period is to have the possibility of specifying timeouts for the amount of time to wait for messages. This is a basic mechanism on which higher level failure handling can be built.

3. PERIPHERAL ASPECTS

Some other Termite features are also notable. While they aren't 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, the naming of computers and establishing network connections to them, and migrating computation in the distributed network.

3.1 Serialization

There should not be 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 that 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, in Termite, ports (like open files) or references to physical devices (like the mouse and keyboard) are exposed as processes.

Abstracting non-serializable objects as processes has two other benefits. First, it enables the creation of interesting

[1]Joe Armstrong calls this "send and pray" semantics [2].

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 cause it to be printed, etc. Secondly, this allows non-movable resources to be accessed through the network.

3.2 Explicit mutation

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

The model therefore forbids explicit mutation in the system (as with the special form `set!` and procedures `set-car!`, `vector-set!`, etc.) The fact that no explicit mutation (in the Scheme sense) is allowed in Termite isn't 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 cause the handler to be invoked with the exception as a parameter. The handler can either choose to manage that exceptional condition or to raise it again. If it manages the exceptional condition it can resume execution either at the point the exception was signaled (if the form `handle` is used) or at the point that the handler was installed (if the form `catch` is used). If it raises the exception again, it will cause the nearest encapsulating handler to be invoked. If an exception propagates to the root of the process without being managed (an uncaught exception), the process dies.

Exception propagation between nodes occurs when a process dies and it is linked to other 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, in unreliable. It would be reasonable to think that the implementation might make an extra effort to ensure that an exception is delivered since that kind of message might be more important for the correct execution of the application.

Receiving an exception causes that exception 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, a bidirectional link will be used since both the supervisor and the worker needs to learn about the death of the other (the supervisor so it may restart the worker, the worker so it can stop executing if not supervised anymore). In a monitor-worker relation where the monitor is an exterior observer to the worker, an outbound link from the worker will be used 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 there is a need to, 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 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` operator.

The distributed system is said to be "open": nodes can be added or removed from a distributed computation at any time.

As it is possible to spawn a process on the current node, it is possible to spawn a process on a remote node by migrating a process. This is one of the key features that enable distribution.

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

3.5 Remote procedure calls

A process might make multiple concurrent requests to another process. Also, replies to requests might come from a third party. In those cases, it must be possible to uniquely mark the requests in order to be able to identify which reply goes with which request. For that we use universally unique references called *tags*. They can be used to distinguish data at the whole distributed network level.

3.6 Process migration

Processes aren't necessarily tied to the node they are currently on. A process that isn't acting as a representative of a non-serializable object is independent of the location where it is executing. This opens up the possibility of migrating processes between nodes. Process migration may be useful in some circumstances: it may be used to implement load-balancing (migrate process to systems with a lower load), move a running program to another computer in order to perform maintenance, migrate the code execution instead of resources in cases where it is less costly to do so, etc.

Process migration is implemented by serializing a continuation. Since a continuation is somewhat like a representation of a process frozen at some point in time, transmitting it through the network enables process mobility.

Process migration, contrary to task migration, implies that the process keeps its identity once it has moved to another node. This also means that for the other processes, a process migration is transparent: they can keep sending it messages and the runtime will ensure that those will follow to the right destination.

4. THE TERMITE LANGUAGE

This section introduces the Termite language through examples. It is important to note that for the sake of simplicity those examples assume that messages will always be delivered (no failure) and that they will be received in the same order that they are sent.

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:

```
(define pong-server
  (spawn
    (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 `?` operator 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 certain criteria.

The selective message retrieval operator is `??`. It takes a predicate for argument. The first message in the mailbox which satisfies that predicate will be retrieved.

Here’s 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 verbose. Since those are frequent operations, Termite offers a pattern matching facility.

Pattern matching is implemented as a macro 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
    (let loop ()
      (recv
        ((from 'ping)        ; pattern to match
         (where (pid? from)) ; additional constraint
         (! from 'pong)))    ; action
      (loop))))
```

4.4 Using timeouts

The way to deal with unreliable message delivery is to specify a maximum amount of time to wait for the reception of a message. This is achieved by giving a timeout delay and a default value to be returned if the timeout is reached as optional arguments to the message retrieval procedures (ie. `?` and `??`). If no timeout is specified, the process will wait forever. If no default value is specified, the `timeout` symbol will be raised as an exception. It is also possible to add an extra `after` clause to `recv`:

```
(! 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

Here is an example of an RPC server to which we make uniquely identified requests. In this case a synchronous call to the server is made:

```
(define rpc-server
  (spawn
    (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 value arguments like the message retrieving procedures.

4.6 Mutable data structure

Mutation isn't allowed in Termite in the same way that it is present in Scheme, but it is still possible to implement mutable data structures using a suspended process to represent state. Here is an example of the implementation of a cell:

```
(define (make-cell content)
  (spawn
    (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 (like an error) is done using the `raise` procedure. Exception handling is done using one of the special forms `catch` and `handle`, which installs a dynamically scoped exception handler. The procedure `spawn-link` creates a new process, just like `spawn`, but that new process is bidirectionally linked with the current process.

After invoking the handler on an exception, the form `catch` will resume execution at the point where the handler was installed, while the form `handle` will resume execution at the point where the exception was raised. The following example illustrates this difference:

```
(catch
 (lambda (exception) exception)
 (raise 42) ; this will not return
 123)                               ⟹ 42

(handle
 (lambda (exception) exception)
 (raise 42) ; control will resume here
 123)                               ⟹ 123
```

This example shows how an exception can propagate through a link between two processes:

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

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

4.8 Process migration

To migrate a process to a remote node a reference to the remote node is needed. It is constructed by giving the IP and TCP port number as arguments to the `make-node` procedure. Through its migration a process keeps its identity and the content of its mailbox. Here is an example of its use:

```
;; reference to the current node
(define node1 (current-node))
;; reference to some remote node
(define node2 (make-node "example.com" 3000))

(define migrating
  (spawn
    (let loop ()
      (recv
        ((from tag 'where)
         (! from (list tag (current-node))))

        (('migrate to)
         (migrate to))))))

(!? migrating 'where)               ⟹ node1
(! migrating node2)
(!? migrating 'where)               ⟹ node2
(! migrating node1)
(!? migrating 'where)               ⟹ node1
```

Note that the procedure `migrate` has to be called by the process that is migrating. A process cannot force another to migrate, it needs the collaboration from the migrating process. This is important because in general a change in a process' execution context requires it to adapt to its new location.

4.9 Remotely spawning a process

In order to spawn a process remotely the process is first spawned on the local node, and then migrates to the designated node:

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

(let ((me (self)))
  (spawn
    (migrate node)
    (! me 'boo)))                   ⟹ a pid

(?)                                 ⟹ boo
```

Note that establishing links to remote processes will also work:

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

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

4.10 Abstractions built using continuations

It is possible to define interesting abstractions by using `call/cc`. In this section we give as an example process cloning and dynamic code update.

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

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

Dynamic code update in a running system can be interesting, especially with long-running computations or high-availability environments. Here is an example of a running process having its code being updated dynamically:

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

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

(define new-server
  (spawn
    (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

(let ((replacement (!? new-server 'clone)))
  (! server (list 'update replacement)))

(!? 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. Note that this necessitates cooperation from the process having its code replaced (it understands the `update` message).

5. EXTENDED 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 will supervise 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 such a supervisor might be started:

```
(define (start-meter-supervisor)
  (spawn
    (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 "work dispatching server":

```
(define (start-work-dispatcher load-server)
  (spawn
    (let loop ()
      (recv
        ((from tag ('dispatch thunk))
         (let ((min-loaded-node
                (minimum-load load-server)))
           (spawn
             (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 a thunk to be executed on the most lightly loaded node in a cluster.

5.2 Abstracting Concurrency

Since building distributed applications is a complex task, it is a good practice to abstract the notion of concurrency when expressing common patterns. An example of such a common pattern would be a server process that will be used in a classic client-server pattern. Erlang's concept of behaviors is used 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 will answer 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 to be used as callbacks when certain events occur in the server.

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

A generic server plugin contains four closures, each to be called to react to a particular situation. The first is for server initialization, called when the server is created. The second is for procedure call to the server: the closure will dispatch 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 a function to be called when the server is terminated.

Here's 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 will help in building reliable software, because the software development process will be less error-prone. We reduce complexity at the cost of losing some flexibility.

5.3 Fault Tolerance

Encouraging simpler code is only a first step in order to be able to build robust applications. We also need to be able to handle system failures and software errors. Supervisors are another kind of behaviors in the Erlang language, but we give a slightly different implementation from Erlang's here. A *supervisor* process is responsible for supervising the correct execution of a *worker* process. If there is a failure in the worker, the supervisor will restart 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 will generate 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 *pid* returned by the call to `spawn-thunk-supervised` is the one of the supervisor, but any message sent to the supervisor will be sent to the worker. The supervisor is then mostly transparent: interacting processes don't necessarly know that it is there.

There is one special message that will be intercepted by the supervisor, and that is a message consisting of the single symbol `'shutdown`. Sending that message to the supervisor will make it invoke a `shutdown` procedure that will request the process to end its execution, or terminate it if it doesn't collaborate. In the previous trace, the "had to terminate the process" message indicates that the process didn't 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 will be 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 they all be restarted when a single process fails or just the failed process, etc.).

6. THE TERMITE IMPLEMENTATION

A prototype Termite system has been implemented. It is built on top of Gambit-C 4 [5]. Two features of Gambit-C were very helpful when implementing the system: lightweight threads and object serialization.

Gambit-C supports lightweight language-level threads as specified by SRFI-18 [6] and SRFI-21 [7]. It is possible to start millions of threads in a single program. This makes the Termite model applicable when used with Gambit.

Serialization is also supported for an interesting subset of Gambit-C objects. In particular closure and continuation serialization is supported. This makes it possible to implement process migration in the language by using `call/cc`.

Location transparency means that messages sent to a process will reach it even if it has migrated from its original position. This is transparent for the sending process. It is implemented by keeping a *migration table* on each node. This table keeps track of the location of processes that were once executing on the node but have migrated away. Each process has *thread-local* information that stores the set of nodes on which the process has been executing. When a node receives a message that is intended for a process that has migrated away, the node forwards the message to the node where the process has migrated and sends back a message to the node from which the message originates in order to allow it to update its migration table. This is a technique derived from the one explained in [13]. When a migrating process stops executing, it sends a message to each of the nodes it has executed on so they can clean up their migration table.

No modification of the underlying Gambit system was needed to implement the prototype Termite system, but it ended up being a motivation to improve the performance of Gambit's serialization mechanism.

7. EXPERIMENTAL RESULTS

In order to evaluate the performance of Termite, some benchmark programs were executed using Termite version 0.4. When possible, the equivalent Erlang program was executed using Erlang/OTP version 5.4.8 to compare the two systems. Moreover, some of the benchmarks were also rewritten directly in Gambit Scheme and executed with version 4.0 beta 14 to evaluate the implementation overhead. All the benchmarks were executed on 2.2 GHz AMD Athlon 64-based PCs with 2GB RAM and a 100Mb/s Ethernet running Linux 2.6.10.

7.1 Ring of processes

This benchmark creates a ring of 250,000 processes on a single node, then sends around the ring a number that is decremented by each process successively. When the number passed around is 0, the process stops executing after relaying the 0 to the next process.

The benchmark is run two times: first, with an initial seed of 0, so that all that is done is to create the ring then pass a message around to destroy it, in order to put more strain on process creation and destruction. Then it is run with an initial seed of 1,000,000, so that message passing time becomes more significant in the test. Performance is given in seconds, lower is better.

Seed	Erlang	Gambit	Termite
0	0.92	2.82	4.88
1,000,000	1.63	5.53	8.21

7.2 Ping-Pong Exchanges

This benchmark measures the maximum number of "ping-pong" exchanges per second that can be made between two processes, under three different conditions: when the processes are executing on the same node, when the processes are executing on two nodes running on the same computer and finally when the processes are executing on two nodes running on different computers but on the same local network. Performance is given in number of round-trips per second, higher is better.

	Erlang	Gambit	Termite
Same node	2,001,366	457,415	206,079
Same computer	16,603	–	1,662
Local network	10,970	–	1,037

7.3 Process Migration

This benchmark was made with Termite only, it measures the process migration time for between two nodes on the same computer and between two nodes running on different computers but on the same local network. Performance is given in seconds, lower is better.

	Termite
Same computer	0.109
Local network	0.115

Note that the Termite code was compiled for the two first benchmarks, while it was interpreted for the migration benchmark. This is due to current implementation issues related to the serialization of compiled code. The Scheme and the Erlang code were compiled for all benchmarks.

8. RELATED WORK

The **Actors** model is a general model of concurrency that has been developed by Hewitt, Baker and Agha [11] [10] [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 doesn't 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 has been developed in the context of building telephony applications, which are inherently concurrent. The idea of multiple lightweight isolated processes with unreliable asynchronous mesrsage 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 some regards. Those characteristics have motivated the idea of integrating Erlang's concurrency ideas to a Lisp language. Termite notably adds to Erlang first-class continuations, macros and the possibility to transparently migrate processes. 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 doesn't 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 [9] 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 intends to be a minimal addition to Scheme that enables distributed programming. Unlike Termite, it doesn't feature lightweight isolated process and doesn't consider the problems associated with failures.

Dreme [8] 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 if objects are not explicitly migrated, sending them to remote processes will be done by reference. 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

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

The current implementation is built on the Gambit Scheme system. While this has had 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 constant need for serialization, try to have processes as lightweight and efficient as possible, look into optimization at the level of what needs to be transferred between nodes, etc. Apart from the optimization 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 article.

10. ACKNOWLEDGMENTS

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

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

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

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

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

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

[10] 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.

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

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

[13] WooYoung Kim and Gul Agha. Efficient support of location transparency in concurrent object-oriented programming languages. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 39, 1995.