

Termite – A Lisp for Distributed Computing

Presented by Guillaume Germain

Work done in collaboration with Marc Feeley and Stefan Monnier

Université de Montréal

Presentation Plan

- Goals
- Model
- Termite
- Implementation
- Future work

Goals and Needs

Goal: to have a good language to develop distributed applications and experiment with distributed computing patterns.

Needs:

- A good concurrency/distribution model.
- A simple, flexible language good for building abstractions.

The solution: a blend of Erlang and Scheme.

Model – Erlang

- Message-passing model of concurrency,
- Isolated processes,
- No mutation,
- Each process has one mailbox,
- Messages are sent asynchronously,
- Messages are retrieved synchronously,
- Sending a message is an unreliable operation,
- Lightweight processes.

Model – Scheme

Scheme's clean and simple model

- Higher-order procedures,
- First-class continuations,
- Macros.

Termite – Basic Operators

- Starting a new process:
`(spawn . <body>)`
- Get the *pid* of the current process:
`(self)`
- Sending a message:
`(! pid message)`
- Retrieving the first message in the mailbox:
`(? [timeout [default]])`
- Retrieving the first message satisfying a predicate:
`(?? pred [timeout [default]])`

Termite – Pattern Matching

recv: retrieving a message by pattern matching

```
(define square-server
  (spawn
    (let loop ()
      (recv
        ((from x)
         (! from (* x x))))
      (loop))))

(! square-server (list (self) 5))
(?)
```

Termite – Tags

Handling multiple concurrent requests using *tags*

```
(define square-server
  (spawn
    (let loop ()
      (recv
        ((from tag x)
         (! from (list tag (* x x))))))
      (loop))))

(let ((tag (make-tag)))
  (! square-server (list (self) tag 5))
  (recv (,tag reply) reply))
```

Termite – RPC

Abstracting a common pattern

```
(define square-server
  (spawn
    (let loop ()
      (recv
        ((from tag x)
         (! from (list tag (* x x))))))
      (loop))))

(!? square-server 5)
```

Termite – Migration

Transparent process migration

- Identity is preserved,
- Messages follow-up automatically.

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

```
(define server@node  
  (spawn  
    (migrate node)  
    (start-server)))
```

Termite – Abstractions

Using processes as abstractions for:

- Mutable data structures,
- Ports (in the Scheme sense),
- Physical devices (keyboard, mouse, etc.),
- etc.

Termite – Continuations

Uses of `call/cc`

- Process migration and cloning,
- Code update,
- State rollback,
- etc.

```
(define (migrate-task to)
  (call/cc
    (lambda (k)
      (! to k)
      (halt!))))
```

Example: Futures

```
(define-macro (future . body)
  '(future* (lambda () ,@body)))
```

```
(define (future* thunk)
  (spawn
    (let ((value (thunk)))
      (let loop ()
        (recv
          ((from tag 'ref)
           (! from (list tag value))))
        (loop))))))
```

```
(define (touch promise)
  (!? promise 'ref))
```

Implementation

Built on top of Gambit-C 4.0

Good:

- Fast and lightweight threads,
- Serialization of closures and continuations,
- Not much code.

Not so good:

- Global environment not included in serialization,
- Serialized closures are big,
- Compiled code issues.

Future work

Short term

- Refine the concept of nodes,
- Improve security.

Medium/Long term

- Robust and efficient implementation from scratch,
- A good graphical interaction and development environment.

Conclusion

Termite is an appropriate and interesting language and system to implement distributed programs. It is simple yet flexible, and allows for the abstraction of patterns of distributed computation.

The End

- Questions,
- Comments,
- Suggestions...