NeXeme
A Distributed Scheme based on Nexus
Reference Manual and User's Guide

Luc Moreau[1]
Technical Report M97/8
University of Southampton

November 97

[1] Id: ug.tex,v 1.1 1997/11/09 16:07:05 lavm Exp lavm .

Abstract

The remote service request, a form of remote procedure call, and the global pointer, a global naming mechanism, are two features at the heart of Nexus, a library for building distributed systems. NeXeme is an extension of Scheme that fully integrates both concepts in a mostly-functional framework, hence providing an expressive language for distributed computing. This document is both NeXeme reference manual and user's guide.

Contents

1 Introduction

NeXeme [3] is a distributed implementation of Scheme using the library for distribution Nexus, the thread library PPCR, and the Bigloo Scheme compiler.

1.1 Nexus

Nexus [1] is structured in terms of five basic abstractions: nodes, contexts, threads, global pointers, and remote service requests. A computation executes on a set of *nodes* and consists of a set of *threads*, each executing in an address space called a *context*. (For the purposes of this article, it suffices to assume that a context is equivalent to a process.) An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same context.

The *global pointer* (GP) provides a global name space for objects, while the *remote service request* (RSR) is used to initiate communication and invoke remote computation. A GP represents a communication endpoint: that is, it specifies a destination to which a communication operation can be directed by an RSR. GPs can be created dynamically; once created, a GP can be communicated between nodes by including it in an RSR. A GP can be thought of as a capability granting rights to operate on the associated endpoint.

Practically, an RSR is specified by providing a global pointer, a handler identifier, and a data buffer, in which data are serialised. Issuing an RSR causes the data buffer to be transfered to the context designated by the global pointer, after which the routine specified by the handler is executed, potentially in a new thread of control. Both the data buffer and pointed specific data are available to the RSR handler.

The remote service request mechanism allows point-to-point communication, remote memory access, and streaming protocols to be supported within a single framework.

1.2 Overview of the Implementation

Bigloo [4] is a Scheme compiler that generates C intermediate code. This Scheme implementation was adopted to develop NeXeme as it facilitates the integration of C libraries with Scheme.

Figure 1 displays the organisation of the NeXeme implementation. As Nexus is multithreaded, we had to adopt a thread-safe garbage collector. The public domain Boehm-Weiser's [2] garbage collector supports various OS-level threads and is also part of the PPCR portable common runtime system [5]. On each platform, Nexus is recompiled against the garbage collector and the thread package. The executable is generated by linking the Scheme system with the resulting libraries Nexus, gc, and threads.

Primitives of the Nexus library are made accessible through a foreign interface definition. Let us note that some Nexus primitives require procedures as argument (for instance, thread creation or callbacks for handlers). In order to integrate properly Nexus with Scheme, we modified Nexus to support callbacks to Scheme functions. While the foreign interface defines a similar API as the Nexus library, the "functional Nexus" layer

Utilities		
NeXeme		
Functional Nexus		
Foreign Interface		
Nexus	Threads	Scheme
GC		

Figure 1: Organisation of NeXeme

provides a more functional interface to Nexus; for instance, results are returned by functions, errors are signalled by exceptions, and memory is managed automatically. The NeXeme layer offers a functional version of remote service requests as described in [3]. Finally, a library defining a set of utilities provides other paradigms for distribution like futures, communication channels, or farm processing.

2 NeXeme Primitives

As NeXeme is a language defined on top of Nexus, many NeXeme primitives inherit their behaviour from the corresponding Nexus primitives. The reader is invited to read Nexus reference manual and user's guide for more information.

2.1 Initialisation

◇ `(nexeme-init)` (module: toplevel)

Nexus initialisation is quite complex because numerous parameters may be passed as argument. A high-level procedure procedure for initialisation `nexeme-init` is provided. It returns a list containing the current node. Note that `nexeme-init` initialises Nexus but also all NeXeme packages (including the distributed garbage collector). In addition, all remote service request handlers defined by the user with `define-rsr-init-handler` will also be installed by the `nexeme-init`.

Note that only the master node returns from a call `nexeme-init` (as it is the case with the Nexus function `nexus_start`).

◇ `(nexus-node? val)` (module: bigloo-nexus)
`(nexus-node->gp val)` (module: bigloo-nexus)
`(nexus-node->name val)` (module: bigloo-nexus)
`(nexus-node->number val)` (module: bigloo-nexus)
`(nexus-node->return_code val)` (module: bigloo-nexus)

A Node is a Nexus data structure composed of four fields; a global pointer, a host name (the node number and return code are two Nexus specific fields).

◇ `(nexus-init ...)` (module: bigloo-nexus)
`(nexus-start)` (module: bigloo-nexus)
`(bigloo-nexus-init ...)` (module: bigloo-nexus)
`(simple-bigloo-nexus-init argv)` (module: bigloo-nexus)

These lower level initialisation procedures are also available, but we recommend to cautiously use them as NeXeme packages will not be initialised and no distributed garbage collector will be available.

2.2 Remote Context Startup

A remote context may be started up by a call to `nexus-acquire-nodes`. The calling context is said to be *master*, whereas the created contexts are said to be slaves.

◇ `(nexus-acquire-nodes node-name . ...)` (module: bigloo-nexus)

A remote context is created by `nexus-acquire-nodes` which takes a mandatory argument: the name of the host on which a new context has to be created. Optional arguments are also accepted in the following order (as in the corresponding Nexus call):

`node-number` For multiprocessors systems; if `node-number` is not specified, then its value is 0 by default.

`count` Number of contexts to be created on remote host. If `count` not specified, then its default value is 1.

`dir-path` Directory where to start remote execution. If not specified, its value is give by the field `startup_dir` for that node in the `.resource_database` file. If `#f`, then directory is chosen as the current one (warning, be sure that the remote file hierarchy is the same as the local one).

`exec-path` The executable to run. If not specified, its value is give by the field `startup_exe` for that node in the `.resource_database` file. If `#f`, then executable is the same as the current one.

There is also a means of establishing a communication between two independent NeXeme entities. One has to decide to listen on a port, and the other has to connect to this port.

◇ `(start-remote-node host cmd dir user)` (module: bigloo-nexus)
`(nexus-database-lookup host node-num key)` (module: bigloo-nexus)

A NeXeme process can attach itself to Nexus process by giving a URL of the shape:

```
x-nexus://theotherhost:port/string1/string2/...
```

The URL specifies the host and the port number. The result of a successful call is a global pointer (actually a remote pointer) pointing at the remote host; if the call fails the returned value is a number indicating the error that occurred.

A Nexus process can decide to list on a port with `nexus-allow-attach`. The second argument is an optional port; if not specified, Nexus selects a port itself. The value returned by a successful call to `nexus-allow-attach` is a pair containing the current hostname and the port number on which clients have to connect.

The first argument of `nexus-allow-attach` must be a function that takes a string as argument and returns a global pointer pointing at a local object. This function receives the URL passed by the client Nexus process: it allows the server to return different global pointers according to the passed URL.

◇ `(nexus-attach url)` (module: entry-exit)
`(nexus-allow-attach f . port)` (module: entry-exit)

`start-remote-node` starts the execution of a command `cmd` on `host`, in directory `dir`, after logging in as `user`.

Finally, the `.resource_database` file contains information about hosts. The function `nexus-database-lookup` is used to retrieve information from this file. It takes a host name, i.e. a string, a number, a node number, and a key, i.e. a string.

2.3 Remote Service Requests

◇ `(rsr handler-string gp arg1 ...)` (special form: pervasive-macros.scm)
`(define-rsr-handler name args . body)` (special form: pervasive-macros.scm)
`(define-rsr-init-handler name args . body)` (special form: pervasive-macros.scm)

A remote service request is sent to an object pointed by a global pointer using the macro `rsr`. It takes the name of the handler that handles the message (`handler-string` must evaluate to a string), the global pointer at which the request is sent, and optional arguments.

Both `define-rsr-init-handler` and `define-rsr-handler` declare handlers for remote service request. The former form may be used before calling `nexeme-init`, which will take care of installing all handlers defined by this form. The latter form can only be used after `nexeme-init` has been called; it can be used to add or update handlers dynamically at runtime. By default, the handlers declared with those forms are *threaded*.

`define-rsr-init-handler` and `define-rsr-handler` take the name of the handler to be declared, i.e. a string, the list of expected arguments; the `body` is a Scheme expression that may refer to these arguments. Note that the first value bound with `arg` is the *object* at which the remote-service-request is aimed at, whereas the following arguments are those passed during the call to `rsr`.

The first subexpression of the body may be of the form `(origin` *var*`)`. As a result the variable *var* will be bound to a global pointer on the origin site.

The form `define-rsr-init-handler` is particularly useful for nodes which are started as slaves: as the call to `nexeme-init` does not return, initial handlers can be defined with `define-rsr-init-handler`.

◇ `(nexus-init-remote-service-requestf gp handler-name handler-id)` (module: bigloo-nexus)
`(nexus-send-remote-service-request buffer)` (module: bigloo-nexus)
`(nexus-handler-hash name)` (module: bigloo-nexus)

Lower level primitives are also available. `nexus-init-remote-service-requestf` takes a global pointer, the name of a handler, and its hash code; it returns a buffer that can be sent with `nexus-send-remote-service-request`. The function `nexus-handler-hash` computes the hash-code of a handler name.

◇ `(nexus-init-register-handler name fct)` (module: bigloo-nexus)
`(nexus-init-register-non-threaded-handler name fct)` (module: bigloo-nexus)
`(nexus-register-handler name type fct)` (module: bigloo-nexus)

`nexus-init-register-handler` and `nexus-init-register-non-threaded-handler` allow the programmer to define a "nexus style" handler, which will be threaded with the former or non-threaded with the latter. Both functions take the name of the handler, i.e. a string, and a binary function; the binary function will be applied to the object receiving the request and the received buffer. Again those functions for registering handlers may be used before calling `nexeme-init`. Handlers may be updated or declared at runtime using `nexus-register-handler`. Its second argument indicates whether the handler should be threaded (`threaded-type`) or non-threaded (`non-threaded-type`).

2.4 Global Pointers

◇ `(nexus-global-pointer-on-scheme-obj obj)` (module: bigloo-nexus)
`(nexus-convert-global-pointer-address gp)` (module: bigloo-nexus)
`(nexus-global-pointer? obj)` (module: bigloo-nexus)
`(nexus-global-pointer-to-current-context? gp)` (module: bigloo-nexus)
`(nexus-same-global-pointer? gp1 gp2)` (module: bigloo-nexus)
`(nexus-same-context? gp1 gp2)` (module: bigloo-nexus)
`(nexus-hash-global-pointer gp)` (module: bigloo-nexus)
`(nexus-global-pointer-string gp)` (module: bigloo-nexus)

The function `nexus-global-pointer-on-scheme-obj` creates a new global pointer on a Scheme object. If a GP points at some data in the current content, it satisfies the predicate `nexus-global-pointer-to-current-context?`. Furthermore, the function `nexus-convert-global-pointer-address` returns the object it is referencing; the behaviour of this function is not specified when the GP is not pointing at a local object. Equality on global pointers is provided by `nexus-same-global-pointer?`. The predicate `nexus-same-context?` indicates whether two GPs point at the same context. A hash function for global pointer is also provided. The function `nexus-global-pointer-string` is used for debugging purpose: it returns a string representing a GP.

Remark. *Nexus global pointers have two different representations in NeXeme: namely,* global pointers *and* remote pointers. *Global pointers are created with* `nexus-global-pointer-on-scheme-obj` *and they point at a* local *object. Remote pointers are nexus global pointers that point at remote objects. Two predicates distinguish them:* `nexus-global-pointer?` *and* `remote-pointer?`. *The user has no primitive to create a remote pointer; such a facility is only available to NeXeme during the (de)serialisation process.*

Furthermore, remote pointers have a unique representation in memory. If `gp` *points at a remote host, the following expression returns true.*

```
(eq? (remote-eval gp (lambda () (nexus-global-pointer-on-scheme-obj 'a)))
     (remote-eval gp (lambda () (nexus-global-pointer-on-scheme-obj 'a))))
```

Uniqueness of global pointers in memory is not guaranteed. The value of the following expression is not specified.

```
(eq? (nexus-global-pointer-on-scheme-obj 'a)
     (nexus-global-pointer-on-scheme-obj 'a))
```

In practice, in the current implementation, the returned result is `#f`, *but it is a bad programming practice to rely on that property.*

Most functions and special forms such as `rsr`, *..., are generic and accept remote pointers and global pointers.*

Warning *The distinction between remote pointers and global pointers might disappear in a future release, but not the property that two GPs pointing at the same remote object have the same unique representation in memory.*

2.5 Threads

The thread mechanism available in NeXeme is directly inherited from Nexus, which itself imports them from `ports0`, a subset of Posix threads.

◇ `(nexus-thread-create thunk)`	(module: bigloo-nexus)
`(nexus-thread-exit)`	(module: bigloo-nexus)
`(nexus-thread-yield)`	(module: bigloo-nexus)
`(nexus-thread-self)`	(module: bigloo-nexus)
`(nexus-thread-equal? t1 t2)`	(module: bigloo-nexus)
`(nexus-thread-key-create)`	(module: bigloo-nexus)
`(nexus-thread-getspecific Nexus-thread-key)`	(module: bigloo-nexus)
`(nexus-thread-setspecific Nexus-thread-key obj)`	(module: bigloo-nexus)

The primitive `nexus-thread-create` creates a new thread that activates the thunk received as argument. Note that there are no equivalents to `pthread_detach` and `pthread_join` in Nexus. All Nexus threads are automatically detached when they are created.

`nexus-thread-exit` terminates the calling thread. `nexus-thread-yield` yields the processor to another thread. `nexus-thread-self` returns the thread ID of the calling thread. `nexus-thread-equal?` compares two thread IDs.

Thread specific operations are also supported. `nexus-thread-key-create` creates a thread-specific data key that is visible to all threads in the context. The same key may be used by different threads, but the values bound to the key by `nexus-thread-setspecific` are maintained on per-thread basis. The value associated with the calling thread may be obtained with `nexus-thread-getspecific`.

> **Warning [Solaris]** In the implementation of Nexus on top of Solaris threads, thread-specific primitives are mapped directly to the corresponding Solaris primitives. As a result, the garbage collector is unable to trace objects bound to thread-specific keys.

> **Warning [PPCR]** In the implementation of Nexus on top of PPCR threads, all thread primitives are mapped directly to the corresponding PPCR primitives. PPCR does not follow POSIX thread semantics. The most noticeable difference concerns file descriptors: when a new thread is created it inherits the file descriptors accessible by its parent, as opposed to file descriptors accessible in the whole process. The semantics is very similar to the semantics of fork and unix process.
>
> A MAJOR and unfortunate consequence is that Scheme ports *cannot be shared between threads*!

2.6 Mutex and Conditional Variables

◇ `(nexus-mutex-create)` (module: bigloo-nexus)
`(nexus-mutex-lock mutex)` (module: bigloo-nexus)
`(nexus-mutex-unlock mutex)` (module: bigloo-nexus)
`(nexus-cond-create)` (module: bigloo-nexus)
`(nexus-cond-wait cond mutex)` (module: bigloo-nexus)
`(nexus-cond-signal cond)` (module: bigloo-nexus)
`(nexus-cond-broadcast cond)` (module: bigloo-nexus)

NeXeme mutex and conditional variables are directly imported from Nexus. As a result NeXeme primitives are the same as for Nexus.

> **Warning [PPCR]** In the implementation of Nexus on top of PPCR threads, all conditional variable primitives are mapped directly to the corresponding PPCR primitives. PPCR does not follow POSIX thread semantics. In particular, a call to `nexus-cond-wait` returns there is no guarantee that the current thread has acquired the current mutex. This problem seems to be overcome with `nexus-cond-wait!!`, but this should be looked at more closely.
>
> ◇ `(nexus-cond-wait!! cond mutex)` (module: bigloo-nexus)

2.7 Outputs

◇ `(nexus-print . args)` (module: bigloo-nexus)
`(nexus-write . args)` (module: bigloo-nexus)

These output primitives display their arguments in a critical section. This ensures that their text is not interleaved with other texts, printed in parallel by other threads. In addition, they display the host on which the program is running, the context, the thread number and the process, which is very convenient in debugging mode.

2.8 NeXeme Utilities

Nexeme comes with a series of higher-level library functions.

2.8.1 Remote Evaluation

◇ `(remote-eval gp thunk)` (module: nexus-utilities)
`(remote-invoke gp thunk)` (module: nexus-utilities)
`farm-remote-invoke` (module: nexus-utilities)
`remote-eval-ack-handler-name` (module: nexus-utilities)

`remote-eval` and `remote-invoke` have the same signature: they accept a global pointer and a thunk. Both start evaluating the thunk remotely in the context pointed by the global pointer. The value returned by `remote-eval` is the value returned by the thunk, whereas `remote-invoke` returns an unspecified value as soon as the remote computation as been initiated.

2.8.2 Communication Channels

◇ `(make-channel)` (module: nexus-utilities)
`(synchronous-send channel val)` (module: nexus-utilities)
`(synchronous-receive channel)` (module: nexus-utilities)

`make-channel` is the constructor for synchronous communication channels. Values can be sent using `synchronous-send` and received via `synchronous-receive`.

2.8.3 Dynamic Binding and Threads

◇ `(with-dynamic-binding symbol val thunk)` (module: nexus-utilities)
`(dynamic-get symbol)` (module: nexus-utilities)
`(dynamic-set! symbol val)` (module: nexus-utilities)
`(thread-create/db thunk)` (module: nexus-utilities)

There is some basic support for dynamic variables. `with-dynamic-binding` dynamically binds `symbol` with `val`, and applies the `thunk` in this dynamic scope of the new binding. The value of a dynamic variable can be retrieved with `dynamic-get` and assigned with `dynamic-set!`. A thread created with `thread-create/db` inherits the dynamic environment of the calling thread (as opposed to a thread created with `nexus-thread-create`).

2.8.4 Miscellanei

◇ `(nexeme-repl args)` (module: toplevel)

The NeXeme read-eval-print loop is available as a library function. As it takes a Bigloo-style list of parameters, it can be used as a "main" function.

◇ `(with-optional-monitor rest fct)` (module: nexus-utilities)
`(host-alive? gp)` (module: nexus-utilities)
`*gc-messages?*` (module: nexus-utilities)

They do exist but they are too primitive or unstable to be described.

2.9 Debugging

◇ `(debug-level` *package level* `...)` (module: toplevel)

Many packages have their own debugging mode, which can be set by a call to `debug-level`. Recognised packages are identified by the symbols `serial`, `procedure` `libc`, `nexus`, `callcc`, `entry-exit`. The debugging level is a number typically between 0 (no debugging information) and 15 (lot of information).

2.10 Low-level Buffer Handling

In NeXeme, most remote service requests are sent with `rsr` and handled by handlers installed with `define-rsr-init-handler` or `define-rsr-handler`. Those primitives take care of serialising and deserialising any Scheme data. Sometimes, one needs to manipulate Nexus buffers directly when the default serialisation protocol is not suitable.

◇ `(nexus-sizeof-int buffer n)` (module: bigloo-nexus)
`(nexus-sizeof-byte buffer n)` (module: bigloo-nexus)
`(nexus-sizeof-float buffer n)` (module: bigloo-nexus)
`(nexus-sizeof-char buffer n)` (module: bigloo-nexus)
`(nexus-sizeof-global-pointer buffer gp)` (module: bigloo-nexus)
`(nexus-put-int buffer val)` (module: bigloo-nexus)
`(nexus-put-float buffer val)` (module: bigloo-nexus)
`(nexus-put-byte buffer val)` (module: bigloo-nexus)
`(nexus-put-string buffer val)` (module: bigloo-nexus)
`(nexus-put-global-pointer buffer val)` (module: bigloo-nexus)
`(nexus-set-buffer-size buffer size . n-elements)` (module: bigloo-nexus)

Nexus functions to set the size of a buffer and to store data in a buffer are accessible from NeXeme. Note that a GP serialised with `nexus-put-global-pointer` will not be taken care of by the distributed GC, and as a result the data it is pointing at may be incorrectly reclaimed.

◇ `(nexus-get-char buffer n)` (module: bigloo-nexus)
`(nexus-get-int buffer)` (module: bigloo-nexus)
`(nexus-get-int* buffer . rest)` (module: bigloo-nexus)
`(nexus-get-float buffer)` (module: bigloo-nexus)
`(nexus-get-byte buffer)` (module: bigloo-nexus)
`(nexus-get-global-pointer buffer)` (module: bigloo-nexus)

When a RSR is handled, Nexus (version 3.0) distinguishes between stashed and unstashed buffers. NeXeme provides generic functions able to handle both types of buffers.

◇ `(nexus-get-stashed-char buffer n)` (module: bigloo-nexus)
`(nexus-get-unstashed-char buffer n)` (module: bigloo-nexus)
`(nexus-get-stashed-int buffer)` (module: bigloo-nexus)
`(nexus-get-unstashed-int buffer)` (module: bigloo-nexus)
`(nexus-get-stashed-float buffer)` (module: bigloo-nexus)
`(nexus-get-unstashed-float buffer)` (module: bigloo-nexus)
`(nexus-get-stashed-byte buffer)` (module: bigloo-nexus)
`(nexus-get-unstashed-byte buffer)` (module: bigloo-nexus)
`(nexus-get-stashed-global-pointer buffer)` (module: bigloo-nexus)
`(nexus-get-unstashed-global-pointer buffer)` (module: bigloo-nexus)

Functions dealing with stashed and unstashed buffer are also accessible.

2.11 Idle Thread

◇ `(nexus-install-idle-function f)` (module: bigloo-nexus)
`(nexus-shutdown-idle-thread)` (module: bigloo-nexus)

One can install a idle thread that is activated only when no other thread is running. The function `nexus-install-idle-function` takes a thunk as argument and creates an idle thread that executes this thunk. The idle thread can be shut down by calling `nexus-shutdown-idle-thread`.

Note that the distributed garbage collector is using the idle thread to detect when distributed garbage collection activities should proceed. If the idle thread is shutdown, no control message for garbage collection will be sent.

2.12 GC Interface

◇ `(start-gc)` (module: bigloo-nexus)
`(nexus-register-finalizer obj f)` (module: bigloo-nexus)
`(nexus-register-disappearing-link pair obj)` (module: bigloo-nexus)
`(fake-pointer obj)` (module: bigloo-nexus)
`(unfake-pointer obj)` (module: bigloo-nexus)
`(nexus-add-root val)` (module: bigloo-nexus)

Some functionalities of Boehm and Weiser's garbage collector are available from NeXeme. A collection can be explicitly activated with `start-gc`. Finalizers may be installed with `nexus-register-finalizer`: when `obj` becomes garbage, the unary function `f` will be called by the finalization process; the unary function receives the object that is finalized (See comments in the file `entry-exit.scm` about the operations that are allowed in the finalization phase; also see `gc.h` about the existence of cycles).

Basic primitives to create "weak pointers" are also available. The function `nexus-register-disappearing-link` registers the car of a pair (`pair`) to be set to `NULL` when an object `obj` becomes garbage. A pointer can be hidden with `fake-pointer` and made visible again with `unfake-pointer`. Finally, an object can be declared to be a root of the GC with `nexus-add-root`.

2.13 Distributed GC Interface

◇ `(show-message-queues)`	(module: entry-exit)
`(show-exit-table)`	(module: entry-exit)
`(show-exit-table-content)`	(module: entry-exit)
`(show-entry-table)`	(module: entry-exit)
`(flush-message-queues)`	(module: entry-exit)

Content of send/receive tables and message queues can be displayed with the above primitives. It is possible to force the sending of messages with `flush-message-queues`.

◇ `(reference-counters-update)` (module: entry-exit)

The function `reference-counters-update` is the thunk passed to the idle thread to periodically deal with updating reference counters.

2.14 Hash Tables

◇ `(make-association-table size)`	(module: tables)
`(put-table! table key value)`	(module: tables)
`(get-table table key)`	(module: tables)
`(remove-table! table key)`	(module: tables)
`(assgp key alist)`	(module: tables)

NeXeme provides a hash table library that supports global pointers. `make-association-table` is a hash table constructor. `put-table!` store a `value` associated with a `key` in a `table`. `get-table` returns the value associated with `key` in a `table`. `remove-table!` removes the entry associated with `key` in the table.

Finally, the function `assgp` is an assoc function with a predicate suitable to compare global pointers.

2.15 Shutdown

◇ `(nexus-destroy-current-context)` (module: bigloo-nexus)
`(shutdown-nodes nodes1 nodes2)` (module: nexus-utilities)
`shutdown-hook` (module: nexus-utilities)

`nexus-destroy-current-context` kills the current context after properly disconnecting with other sites.

`shutdown-nodes` is a (rather unsuccessful) attempt to define a high-level shutdown procedures. It takes a list of GPs whose nodes have to be shut down and a list of GPs whose nodes remain active. It flushes gc messages and prevents further ones to be sent in the future.

2.16 Time and Random Generator Functions

◇ `(random)` (module: bigloo-nexus)
`(seed-random n)` (module: bigloo-nexus)

`seed-random` uses its argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to the function `random`. (They correspond to Unix `srand` and `rand`.)

◇ `(make-seed-location val)` (module: bigloo-nexus)
`(reentrant-random seed*)` (module: bigloo-nexus)

In the presence of threads, one prefers to use `reentrant-random` which is a reentrant variant of `random`; it takes a "pointer to a seed" created by `make-seed-location`.

◇ `(get-time)` (module: bigloo-nexus)
`(diff-times t1 t2)` (module: bigloo-nexus)
`(diff-utime t1 t2)` (module: bigloo-nexus)
`(diff-stime t1 t2)` (module: bigloo-nexus)
`(clock->sec x)` (module: bigloo-nexus)

`get-time` returns the information contained in a Unix `time` structure as a list with four elements: `tms_utime` is the CPU time used while executing instructions in the user space of the calling process; `tms_stime` is the CPU time used by the system on behalf of the calling process; `tms_cutime` is the sum of the `tms_utime` and the `tms_cutime` of the child processes; `tms_cstime` is the sum of the `tms_stime` and the `tms_cstime` of the child processes. `diff-utime` and `diff-stime` computes the different of user time and system time respectively.

2.17 Fault Handling

Nexus provides some hook to deal with faulty situation. Those hooks are available from NeXeme, but a higher-level interface is being researched.

◇ (nexus-enable-fault-tolerance obj) (module: bigloo-nexus)
nexus-fault-none (module: bigloo-nexus)
(nexus-errno num) (module: bigloo-nexus)
nexus-fault-process-died (module: bigloo-nexus)
nexus-fault-process-shutdown-abnormally (module: bigloo-nexus)
nexus-fault-process-shutdown-normally (module: bigloo-nexus)
nexus-fault-attacher-died (module: bigloo-nexus)
nexus-fault-connect-failed (module: bigloo-nexus)
nexus-fault-bad-protocol (module: bigloo-nexus)
(nexus-signal symbol val) (module: bigloo-nexus)
(install-sigint-handler) (module: bigloo-nexus)

2.18 Sockets

Scheme does not provide many primitives to interact with its environment. As a result, some basic C functions related to the socket interface and low-level reading of files are provided.

(net-connect host port) (module: bigloo-nexus)
(net-setup-listener port . backlog) (module: bigloo-nexus)
(net-setup-anon-listener . backlog) (module: bigloo-nexus)
(net-accept sock) (module: bigloo-nexus)
(net-get-sender sock) (module: bigloo-nexus)
(string-to-display-obj obj) (module: bigloo-nexus)
(close fd) (module: bigloo-nexus)
(close-socket s) (module: bigloo-nexus)
(c-read-line port) (module: bigloo-nexus)
(c-read-char port) (module: bigloo-nexus)
(make-output-port-from-socket fd) (module: bigloo-nexus)
(make-input-port-from-socket fd) (module: bigloo-nexus)

2.19 Mobile Objects

In preparation

2.20 Quantum

In preparation

3 Executing NeXeme

NeXeme comes as an interpreter `nexeme` and a compiler `nexemec`. In this section, we describe the interpreter.

3.1 NeXeme Specific Options

The following flags are recognised by NeXeme:

`-help` displays the various NeXeme-specific options.

`-debug` results in lot of debugging information being displayed.

`-test` calls Christian Queinnec's tester mode (module: tester.scm). This flags must be followed by one or more filenames. Each file name should be composed of a series of Scheme expressions followed by their answer,or `---` if there is no need to test the returned answer. Each expression is evaluated in turn, and its result compared with the immediately following expression. Evaluation stops when a result differs from the indicated one.

3.2 Nexus Specific Options

All Nexus parameters are accepted by NeXeme and are defined in the Nexus User's guide. Nexus parameters should appear between a `-nexus` and `-nexus-end` flags. The following parameters are frequently used by NeXeme programmers:

`-Dnexus` must be followed by the Nexus debugging level.

`-nonameexpand` prevents Nexus to expand directory names when creating a remote context. Should be used cautiously in conjunction with `.resource_database`.

`-debug_command` followed by a command to be executed when a remote context is created. For instance, the file `debugger` launches a remote NeXeme under the control of gdb.

`-debug_display` followed by the X-display on which an xterm must be launched to display the execution of a remote nexus.

A typical execution in tester mode of the file `test/mob.tst` is as follows, with `kronos:0` the X-display.

```
nexeme -nogc_incremental -tests test/mob.tst -nexus  -Dnexus 1 \
  -nonameexpand -debug_command debugger -debug_display kronos:0.0 -nexus_end
```

The command `debugger` is part of the NeXeme distribution, which also contains a special `.gdbinit` file.

The `.resource_database` should contain an entry for each host on which NeXeme is going to run. For instance, the host `kronos` is described as follows:

```
kronos\
   protocols=tcp \
   startups=rsh,ss \
   tcp_interface=kronos \
   domain=.ecs.soton.ac.uk \
   startup_dir=/home/lavm/nexeme/nexus-scheme/
```

3.3 PPCR Specific Options

The flag `-nogc_incremental` disallows the incremental mode of the Boehm and Weiser's GC.

4 Compiling or Interpreting a NeXeme Program

Figure 2 displays the code of a broker program. Two handlers `"register"` and `"find-service"` are defined. Other programs can register a service, described by a string and represented by a global pointer. And other entities can find the existence of a service.

In order to compile the file broker.scm, we just have to call NeXeme compiler `nexemec` as follows:

```
nexemec -o broker broker.scm
```

An object file can simply be generated with the following command.

```
nexemec -c -o broker.o broker.scm
```

It is also possible to interpret a program by using `nexeme` directly. We can then evaluate `(load "broker.scm")` in the read-eval-print loop. However, this requires to pre-load the file `"/u/ur/moreau/nexeme/nexus-scheme/pervasive-macros.scm"`.

5 Known Problems

NeXeme currently relies on a user-level thread package PPCR. As a result, the thread scheduler, i.e. an infinite loop!, is part of NeXeme. This gives this illusion that 90% of the CPU is used, even though no scheme code is actually running.

At the moment, no handler for control-C or control-Z is installed. The only way to kill nexeme is to exit properly with bigloo `exit` procedure, or to use the command `KILL -KILL pid`.

If NeXeme does not exit properly, a process forked by PPCR is not deleted. Also, the parent shell seems to become crazy: any help to explain me this problem is welcome.

6 Interaction with Other Languages

Appendix A contains a Java program communicating with the broker described in Section 4: it registers a translation service to the broker and sits, ready to serve clients calling its "translator" service. Appendix B contains a C program communication with the broker and translator: it connects to the broker to find a translation service, and sends a translation request to the translator.

The size of the code differs quite substantially between the NeXeme version and the other two. The essential reason is that serialisation and deserialisation is handled by NeXeme primitives whereas it has to be explicit in C or Java.

External applications have to conform with the serialisation protocol adopted by NeXeme. Every remote service request buffer contains the following information:

```
(module main
        (main run))

(define *table* (cons 'table '()))
(define store
  (lambda (table key val)
    (set-cdr! table (cons (cons key val) (cdr table)))))
(define retrieve
  (lambda (table key)
    (cdr (assoc key (cdr table)))))

(define-rsr-init-handler "register" (object name agent-gp)
  (nexus-print "In register " name agent-gp)
  (store *table* name agent-gp))

(define-rsr-init-handler "find-service" (object name reply-gp)
  (nexus-print "In find-service " name reply-gp)
  (let ((value (retrieve *table* name)))
    (rsr "ack" reply-gp value)))

(define run
  (lambda (args)
    (nexeme-init)
    (nexus-allow-attach (lambda (url)
                          (nexus-print "URL is " url)
                          (nexus-global-pointer-on-scheme-obj '()))
                        4000)
    (repl)    ))
```

Figure 2: The file `broker.scm`

- an object that refers at the host, source of the RSR (typically this is a GP);
- the arguments of the RSR in a left-to-right order.

Every data serialised in the buffer is preceded by a tag that indicates its type. A tag is encoded as a byte. The types recognised by NeXeme are displayed in Figure 3.

Tag Name	Tag (byte)	Followed by ...
symbol	0	length (int) + string (string)
string	1	length (int) + sequence of characters (string)
nil	2	—
int	3	the integer (int)
float	4	the float (float)
eof	5	—
structure	6	type + number of fields (int) + fields
cons	7	car field + cdr field
struct	8	
global-pointer	9	the global pointer (GP)
remote-pointer	10	the global pointer (GP)
true	11	—
false	12	—
failed	13	—
vector	14	number of fields (int) + fields
proc	15	*architecture dependent*
undefined	16	—
c-zero	17	—
unspecified	18	—
nexus-node	19	name + number + gp + return-code
already-met	20	the number of the object (int)

Figure 3: NeXeme tags

NeXeme preserves sharing of data passed by a RSR. We preserve the unique representation of strings, symbols, pairs, global and remote pointers, structures, vectors, procedures, nodes across a RSR. Each of these objects is (implicitly) given a unique number as it is serialised in the buffer. The tag already-met is followed by a integer that indicates which of these objects have already serialised.

There is subtle distinction between a global pointer and a remote pointer. Both refer to Nexus GPs. However, a Nexus GP is said to be a global pointer if it was allocated on the site that sends the RSR, whereas it is said to be a remote pointer when it points at a site different from the emitter.

A Java Code: the Translator

```
import nexus.*;

public class TranslatorAgent

{
  private Nexus         nexus;
  private char GP_TAG = 9;
  private char STRING_TAG = 1;

  AttachFailedException attach_problem = new AttachFailedException("Attach Failed Alert!");

  public static void main (String args[])
       throws BufferOverrunException, LostPrecisionException, AttachFailedException
  {
    TranslatorAgent agent = new TranslatorAgent();
    agent.start(args);
  }

  public void start(String args[])
       throws BufferOverrunException, LostPrecisionException, AttachFailedException
  {
    GlobalPointer broker_gp, my_gp;

    nexus = new Nexus();
    args = nexus.init(args, "nx", null);

    System.out.println("Starting Client(): calling attach_broker()");

    broker_gp=attach_broker("nexus://kronos:4000/hello/babe");

    my_gp = nexus.global_pointer(this);

    Translator translator= new Translator(nexus, this);

    register_service("translator",broker_gp, my_gp);

    wait_for_reply();

    nexus.destroy_current_context(false);
    System.out.println("TranslatorAgent.start(): exiting");
  }


  protected GlobalPointer attach_broker(String attach_broker_url)
       throws AttachFailedException
  {
    AttachReturn attach_return;

    System.out.println("TranslatorAgent.attach_broker(): calling attach()");

    attach_return = nexus.attach(attach_broker_url);

    if (attach_return.status != 0) { throw attach_problem; }

    return(attach_return.gp);
  }

  void register_service (String name, GlobalPointer broker_gp, GlobalPointer my_gp)
       throws BufferOverrunException, LostPrecisionException
  {
    PutBuffer buffer;

    System.out.println("In register_service()");

    buffer=broker_gp.init_remote_service_request("register",869);
    buffer.set_buffer_size(400,-1);

    buffer.put_u_char(GP_TAG);
```

```
    buffer.put_global_pointer(my_gp);

    buffer.put_u_char(STRING_TAG);
    buffer.put_int(name.length());
    buffer.put_char(name.toCharArray(),0,10);

    buffer.put_u_char(GP_TAG);
    buffer.put_global_pointer(my_gp);

    buffer.send_remote_service_request();
  }

  private synchronized void wait_for_reply()
  {
    while (true)
      {
        try
          {
            wait();
          }
        catch (Exception e) { e.printStackTrace(); }
      }
  }
}

class Translator implements HandlerInterface
{
  private Nexus nexus;
  private GlobalPointer gp;

  private char GP_TAG = 9;      // should probably be defined somewhere else!
  private char STRING_TAG = 1;


  public Translator(Nexus nexus, TranslatorAgent agent)
  {
    this.nexus = nexus;
    this.gp=nexus.global_pointer(this);

    register_handlers(this, nexus);
  }

  public void register_handlers(Translator object, Nexus nexus)
  {
    Handler h[] = new Handler[1];
    h[0] = new Handler("translate",
                       974,
                       Handler.NEXUS_HANDLER_TYPE_THREADED,
                       object,
                       0);
    nexus.register_handlers(h);
  }

  String translate(String val) {
    return("bonjour");
  }

  public void translate_handler (Object address, GetBuffer buffer)
       throws LostPrecisionException, BufferOverrunException
  {
    GlobalPointer emit_gp, reply_gp, my_gp;
    char c;
    int len;
    String str;
    PutBuffer reply_buffer;
    String result;

    my_gp=this.gp;
```

```
    c=buffer.get_char ();
    emit_gp=buffer.get_global_pointer ();

    c=buffer.get_char ();
    len=buffer.get_int ();

    char char_array[] = new char[len+1];
    buffer.get_char(char_array,0,len);
    str=new String(char_array,0,len);

    c=buffer.get_char ();
    reply_gp=buffer.get_global_pointer ();

    result=translate(str);

    System.out.println("translate_handler(): translating " + str + " into " + result);

    reply_buffer=reply_gp.init_remote_service_request("ack",303);
    reply_buffer.set_buffer_size(400,-1);

    reply_buffer.put_u_char(GP_TAG);
    reply_buffer.put_global_pointer(my_gp);

    reply_buffer.put_u_char(STRING_TAG);
    reply_buffer.put_int(result.length());
    reply_buffer.put_char(result.toCharArray(),0,result.length());

    reply_buffer.send_remote_service_request();
    System.out.println("translate_handler(): done");

  }



  public void invoke_handler(String name,
                             int id,
                             int local_id,
                             Object address,
                             GetBuffer buffer)
  {
    switch (local_id)
      {
      case 0:
        try
          {
            translate_handler(address, buffer);
          }
        catch  (Exception e) { e.printStackTrace(); }
        break;
      default:
        System.out.println("invoke_handler(): Error: handler not recognized within object");
        break;
      }
  }
}
```

B C Code: the Client

```
#include <unistd.h>
#include <config/PCR_StdDefs.h>
```

```
#include <io/PCR_IO.h>
#include <base/PCR_Base.h>
#include "nexus.h"
#include "stdio.h"

typedef struct {
  nexus_mutex_t mutex;
  void* result;
} ack_monitor_t;

static void ack_handler(ack_monitor_t *address, nexus_stashed_buffer_t *buffer);

#define FIND_SERVICE_HANDLER_HASH 194
#define ACK_HANDLER_HASH 303
#define TRANSLATE_HANDLER_HASH 974
#define SIZE_TO_BE_COMPUTED 4000

static nexus_handler_t client_handlers[] =
{ {"ack", ACK_HANDLER_HASH,
          NEXUS_HANDLER_TYPE_THREADED,
          (nexus_handler_func_t) ack_handler},
  {(char *) NULL, 0,
                  NEXUS_HANDLER_TYPE_NON_THREADED,
                  (nexus_handler_func_t) NULL},
};


static void attach_broker(char *url, nexus_global_pointer_t *gp)
{
    nexus_printf("attach_broker(): entering\n");
    nexus_attach(url, gp);
}

unsigned char gc_tag_val = 9;
unsigned char str_tag_val = 1;
#define GP_TAG (&gc_tag_val)
#define STRING_TAG (&str_tag_val)




void find_service (char *service_name,
                   nexus_global_pointer_t *broker_gp,
                   nexus_global_pointer_t *translator_gp)
{
    nexus_buffer_t buffer;
    nexus_global_pointer_t reply_gp;
    ack_monitor_t ack_barrier;

    ack_barrier.result=translator_gp;

    nexus_printf("find_service(): entering\n");

    nexus_global_pointer(&reply_gp, &ack_barrier);
    nexus_mutex_init(&(ack_barrier.mutex), NULL);
    nexus_mutex_lock(&(ack_barrier.mutex));

    nexus_init_remote_service_request(&buffer, broker_gp,
                                      "find-service",
                                      FIND_SERVICE_HANDLER_HASH);

    nexus_set_buffer_size(&buffer, SIZE_TO_BE_COMPUTED, -1);
    nexus_put_byte(&buffer, GP_TAG, 1);
    nexus_put_global_pointer(&buffer, (&reply_gp), 1);

    {
      int length = strlen(service_name);
      nexus_put_byte(&buffer, STRING_TAG, 1);
```

```
      nexus_put_int(&buffer, &length,1);
      nexus_put_char(&buffer, service_name, length);
    }

    nexus_put_byte(&buffer, GP_TAG, 1);
    nexus_put_global_pointer(&buffer, (&reply_gp), 1);

    nexus_send_remote_service_request(&buffer);

    nexus_mutex_lock(&(ack_barrier.mutex));
}



void call_translator (nexus_global_pointer_t *translator_gp, char * string, char* result)
{
    nexus_buffer_t buffer;
    nexus_global_pointer_t reply_gp;
    ack_monitor_t ack_barrier;

    ack_barrier.result=result;

    nexus_printf("call_translator(): entering\n");

    nexus_global_pointer(&reply_gp, &ack_barrier);
    nexus_mutex_init(&(ack_barrier.mutex), NULL);
    nexus_mutex_lock(&(ack_barrier.mutex));

    nexus_init_remote_service_request(&buffer, translator_gp,
                                             "translate",
                                             TRANSLATE_HANDLER_HASH);

    nexus_set_buffer_size(&buffer, 4000, -1);
    nexus_put_byte(&buffer, GP_TAG, 1);
    nexus_put_global_pointer(&buffer, (&reply_gp), 1);

    {
      int length = strlen(string);
      nexus_put_byte(&buffer, STRING_TAG, 1);
      nexus_put_int(&buffer, &length,1);
      nexus_put_char(&buffer, string, length);
    }

    nexus_put_byte(&buffer, GP_TAG, 1);
    nexus_put_global_pointer(&buffer, (&reply_gp), 1);

    nexus_send_remote_service_request(&buffer);

    nexus_mutex_lock(&(ack_barrier.mutex));
}


static void ack_handler(ack_monitor_t *address, nexus_stashed_buffer_t *buffer)
{
    nexus_global_pointer_t reply_gp, emit_gp;
    nexus_buffer_t reply_buffer;
    unsigned char discard_byte;


    nexus_printf("ack_handler(): entering\n");

    nexus_get_stashed_byte(buffer, &discard_byte,1);
    nexus_get_stashed_global_pointer(buffer, &emit_gp,1);

    nexus_get_stashed_byte(buffer, &discard_byte,1);

    if (discard_byte==str_tag_val) {
      int tmp;
```

```
      char *result=address->result;

      nexus_get_stashed_int(buffer, &tmp, 1);
      nexus_get_stashed_char(buffer, result, tmp);
      result[tmp]='\0';
    }
    else {
      nexus_global_pointer_t* result=address->result;
      nexus_get_stashed_global_pointer(buffer, result,1);
    }
    nexus_mutex_unlock(&(address->mutex));
}



void client_main()
{
  char translation_result[1024];
  nexus_global_pointer_t broker_gp, translator_gp;

  attach_broker("x-nexus://kronos:4000/kronos/foo/hello", &broker_gp);

  find_service("translator",&broker_gp,&translator_gp);

  call_translator(&translator_gp,"hello",translation_result);

  nexus_printf("exiting translator %s\n",translation_result);
  nexus_destroy_current_context(NEXUS_FALSE);
}

/* ****************************************************************************** */
/*                                                                                */
/*                           PPCR & Nexus initialisation                          */
/*                                                                                */
/* ****************************************************************************** */

void test_main(int argc, char **argv)
{
    int my_argc = 0;
    char **my_argv;
    nexus_node_t *nodes;
    int n_nodes;
    int i;


#ifdef WAIT_FOR_DEBUG
    int iwait4debug=1;
#endif

    nexus_init(&my_argc,
               &my_argv,
               "NEXUS_ARGS",
               "nexus",
               NULL,
               NULL,
               NULL,
               NULL,
               &nodes,
               &n_nodes);

#ifdef WAIT_FOR_DEBUG
    nexus_printf("process waiting to be debugged\n");
    while( iwait4debug ) {
      ;
    }
#endif
    nexus_start();
```

```
    client_main();

}

void NexusExit(void) {}

int NexusBoot(void)
{
    nexus_register_handlers(client_handlers);
    return (0);
}

#undef printf

void PCR_main (int argc, const char **argv, void *data)
{
  printf("Entering PCR_main\n");
  printf("argc %d\n",argc);
  test_main (argc,(char **) argv);
  printf("Hi(3)\n");
}

PCR_Base_App runList[] = {
  { "client2", (PCR_Base_AppProc*) &PCR_main, (void*) NIL,NIL,(PCR_Bool) NIL, (PCR_Base_AppProc*) NIL, (void *) NIL },
  { NIL } };


void main (int argc, const char ** argv)
{
  char buffer[100];
  printf("Entering main\n");
  PCR_Base_StartApps(argc,
                     argv,
                     runList,
                     PCR_Bool_true,
                     buffer,
                     100);
  buffer[99]='\0';
  printf("Error %s\n", buffer);
  exit(-1);

}

void Nexus_foreign_invoker(void) {};
void Nexus_foreign_invoker_non_threaded(void) {};
```

References

[1] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[2] H.-J.Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software – Practice and Experience*, 18(9):807–820, 1988.

[3] Luc Moreau, David DeRoure, and Ian Foster. NeXeme: a Distributed Scheme Based on Nexus. In *Third International Europar Conference (EURO-PAR'97)*, volume 1300 of *Lecture Notes in Computer Science*, pages 581–590, Passau, Germany, August 1997. Springer-Verlag.

[4] Manuel Serrano. *Vers une compilation portable et performante des langages fonctionnels.* PhD thesis, Université Paris VI, December 1994.

[5] M. Weiser, A. Demers, and C. Hauser. The Portable Common Runtime Approach to Interoperability. In *ACM Symposium on Operating System Principles*, pages 114–122, December 1989.