NeXeme: A Distributed Scheme based on Nexus

Luc Moreau*, David De Roure*, and Ian Foster**

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 paper presents a semantics for this Scheme extension, and also describes a NeXeme implementation, including its distributed garbage collector.

1 Introduction

Scheme [20] is a mostly-functional language, i.e., it is a fully functional language that also supports imperative notions such as assignments and continuations, for efficiency and expressivity reasons. We believe that a distributed extension of such a language requires a mechanism to invoke functions remotely, so that distribution becomes part of the most fundamental operation of the language. Such an approach is also adopted by languages such as Obliq [2], Java + RMI [10, 22], and Compositional C++ [4], in which methods can be invoked remotely.

Nexus [7], a library for building distributed systems, has two salient features: a remote service request is a form of remote procedure call [1], and global pointers provide for global naming in a distributed environment. By offering a functionality close to remote function invocation, Nexus is a suitable building block for our distributed language. Furthermore, when designing a distributed version of Scheme, our concerns were portability and potential use of high-performance hardware or protocols (e.g., supercomputers, ATM, UDP). Nexus also addresses these concerns as it runs on a variety of platforms and protocols.

NeXeme integrates the Nexus approach, with its remote service requests and global pointers, into a mostly functional language. The result is a novel distributed programming language that offers expressivity, development ease, and automatic memory management (via a distributed garbage collector). NeXeme provides powerful abstractions for controlling distribution while remaining computationally efficient. We believe that NeXeme is an excellent medium for implementing other forms of parallelism such as communication channels [21, 9] and futures [12, 14]. It is also an ideal platform for developing distributed symbolic applications, based for example on distributed mobile agents.

To this end, in Section 3, we present a formal semantics for a simplified version of NeXeme, called Idealised NeXeme. The semantics is operational as it defines a mechanical way of evaluating NeXeme programs on an abstract machine. In Section 4, we describe the NeXeme implementation. We compare and discuss our approach with related work in Section 5. More information on NeXeme is available at the following URL [15].

2 The Nexus Architecture

Nexus [7] 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

* This research was supported in part by EPSRC grant GR/K30733. Authors' address: Department of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, United Kingdom. E-mail: L.Moreau@ecs.soton.ac.uk.

** Author's address: Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. E-mail: foster@mcs.anl.gov.
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 transferred 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.

3 The Essence of Remote Service Requests

In this section, we present an operational semantics for remote service requests. For the sake of clarity and simplicity, we consider a subset of NxXene, called Identical NxXene. Table 1 summarises its different data structures, their representation, their constructors, and the operations permitted on them. (Programming examples may be found in [15].) The semantics was designed also to model non-functional languages like C; for example, full closures and first-class continuations are present in the semantics, but are not required in the actual implementation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Representation</th>
<th>Constructor</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pair</td>
<td>(cons V1 V2)</td>
<td>cons</td>
<td>cdr, cdr</td>
</tr>
<tr>
<td>box</td>
<td>(box α s)</td>
<td>make-ref</td>
<td>deref, set-ref</td>
</tr>
<tr>
<td>lock</td>
<td>(lk α, s)</td>
<td>make-lock</td>
<td>lock, unlock</td>
</tr>
</tbody>
</table>
| global pointer | (gp α, s)   | make-gp, remote | remote service request
| closure   | (cl λx. M)     | λx.M        | apply         |
| continuation | (co κ)       | callcc      | apply         |

Table 1. Data Structures

The state space of the semantics appears in Figure 1. Distributed computations proceed inside a world composed of several sites\(^5\). Each site is identified by a unique name and is composed of a set $T$ of tasks sharing access to a store $B$. A store associates locations with values. Sites communicate by exchanging requests; each site contains both an output queue $O$ of requests to be sent, and an input queue $I$ of requests to be handled. When arriving at a site, requests are dealt with by a handler contained in $H$, a handler table associating names with closures.

A task is an abstraction of a Nexus thread composed of a computational state and a name. A computational state is a configuration of the CEK abstract machine [5] designed to evaluate the sequential subset of the language. Two states are permitted: $Ev(M, ρ, κ)$ represents the evaluation of a term $M$ in the environment $ρ$, with a continuation $κ$; $Ret(V, κ)$ models the return of a value $V$ to a continuation $κ$. In terms of implementation, the term being evaluated is the program counter, the continuation is the control stack, and the environment is the sequence of frames containing bindings.

\(^5\) In the semantics, a site corresponds to the notion of context in Nexus; we adopt a different name because contexts are usually given a different meaning in semantics [5].
\[ W \rightarrow \{m_1, \ldots, m_n\}\]  
\(m \in M \rightarrow \{T(s, I, O)\}\)  
\(t \in \text{Task} \rightarrow \{\langle C, \tau \rangle\}\)  
\(\theta \in \text{Store} \rightarrow \{(\alpha_1, V_1), \ldots, (\alpha_n, V_n)\}\)  
\(s \in S \rightarrow \{s_1, s_2, \ldots\}\)  
\(\tau \in T \rightarrow \{\tau_1, \tau_2, \ldots\}\)  
\(C \in \text{Cost} \rightarrow \{\text{cost}(M, \alpha, \theta)\}\)  
\(\rho \in \text{Env} \rightarrow \{(x_1, V_1), \ldots, (x_n, V_n)\}\)  
\(I \rightarrow \{I_1, \ldots, I_n\}\)  
\(O \rightarrow \{O_1, \ldots, O_n\}\)  
\(T \rightarrow \{t_1, t_2, \ldots\}\)  
\(H \in \text{Hull} \rightarrow \{(str_1 f_1), \ldots, (str_n f_n)\}\)

\[ M \in A \rightarrow \{V_1, \ldots, V_n\}\]  
\(V_1 \in \text{Value} \rightarrow \{x, \lambda x.M, V\}\)  
\(V \in \text{Value} \rightarrow \{f, \text{str}, c, p, l, gp, k, \tau\}\)  
\(x \in \text{Voc} \rightarrow \{x, x', x, 0\}\)  
\(f \in \text{Clo} \rightarrow \{(\lambda x.M, \rho)\}\)  
\(\text{str} \in \text{String} \rightarrow \{a, c\}\)  
\(c \in \text{Const} \rightarrow \{0, 1, \ldots, \text{void}\}\)  
\(g \in \text{BCost} \rightarrow \{\text{true}, \text{false}, \text{nil}, 0, 1, \ldots, \text{void}\}\)  
\(f \in \text{PCost} \rightarrow \{\text{cons}, \text{car}, \text{cdr}, \text{makeref}, \text{defref}, \text{setref}, \text{callcc}, \text{make-s}, \text{remote}, \text{rs}, \text{fork}, \text{define-handler}, \text{shutdown}\}\)  
\(p \in \text{Pair} \rightarrow \{(\text{cons} V V)\}\)  
\(b \in \text{Box} \rightarrow \{(\text{bx} \alpha, s)\}\)  
\(t \in \text{Locks} \rightarrow \{(\text{tx} \alpha, s)\}\)  
\(gp \in \text{Glob} \rightarrow \{(\text{gp} s, \alpha)\}\)  
\(k \in \text{Cont} \rightarrow \{(\text{kon} k)\}\)  
\(c \in \text{CCode} \rightarrow \{\text{init}, (\text{cx} \alpha \rho V \ldots, M \ldots), \ldots\}\)  
\(R \in \text{Req} \rightarrow \{(\text{req} \langle s, sps, so, \ldots\rangle\}\)  
\(s \in \text{SO} \rightarrow \{(\text{serialized objects}\}\)  
\(sgp \in \text{SO} \rightarrow \{(\text{serialized global pointer}\}\)

**Figure 1.** State Space of the CEKDS-machine
\[
\begin{aligned}
\text{Ev}(M_{1}, \ldots, M_{n}, \rho_{s}) & \rightarrow \text{Ev}(\lambda \alpha, M_{1}, \ldots, M_{n}, \rho_{s}) \\
\text{Ev}(\lambda \alpha, M_{1}, \ldots, M_{n}, \rho_{s}) & \rightarrow \text{Ret}(\text{let } \lambda \alpha, M_{1}, \ldots, M_{n}, \rho_{s}) \\
\text{Ev}(\alpha, \rho_{s}) & \rightarrow \text{Ret}(\alpha, \rho_{s}) \\
\text{Ev}(x, \rho_{s}) & \rightarrow \text{Ret}(\rho(x), \rho_{s}) \\
\text{Ret}(V_{s}, (\text{arg } \rho \ (V_{1}, \ldots, M_{1}, \ldots, V_{n}, \ldots))) & \rightarrow \text{Ev}(\lambda \alpha, M_{1}, \ldots, M_{n}, \rho_{s}) \\
\text{Ret}(V_{s}, (\text{arg } \rho \ (V_{1}, \ldots, M_{1}, \ldots, V_{n}, \ldots))) & \rightarrow \text{Ev}(\lambda \alpha, M_{1}, \ldots, M_{n}, \rho_{s}) \\
\text{Ret}(V_{s}, (\text{arg } \rho \ (V_{1}, \ldots, M_{1}, \ldots, V_{n}, \ldots))) & \rightarrow \text{Ev}(\lambda \alpha, M_{1}, \ldots, M_{n}, \rho_{s}) \\
\text{Ret}(V_{s}, (\text{arg } \rho \ (V_{1}, \ldots, M_{1}, \ldots, V_{n}, \ldots))) & \rightarrow \text{Ev}(\lambda \alpha, M_{1}, \ldots, M_{n}, \rho_{s}) \\
\text{Ret}(V_{s}, (\text{arg } \rho \ (V_{1}, \ldots, M_{1}, \ldots, V_{n}, \ldots))) & \rightarrow \text{Ev}(\lambda \alpha, M_{1}, \ldots, M_{n}, \rho_{s}) \\
\text{Ret}(V_{s}, (\text{arg } \rho \ (V_{1}, \ldots, M_{1}, \ldots, V_{n}, \ldots))) & \rightarrow \text{Ev}(\lambda \alpha, M_{1}, \ldots, M_{n}, \rho_{s}) \\
\text{Ret}(V_{s}, (\text{arg } \rho \ (V_{1}, \ldots, M_{1}, \ldots, V_{n}, \ldots))) & \rightarrow \text{Ev}(\lambda \alpha, M_{1}, \ldots, M_{n}, \rho_{s}) \\
\text{Ret}(V_{s}, (\text{arg } \rho \ (V_{1}, \ldots, M_{1}, \ldots, V_{n}, \ldots))) & \rightarrow \text{Ev}(\lambda \alpha, M_{1}, \ldots, M_{n}, \rho_{s}) \\
\text{Ret}(V_{s}, (\text{arg } \rho \ (V_{1}, \ldots, M_{1}, \ldots, V_{n}, \ldots))) & \rightarrow \text{Ev}(\lambda \alpha, M_{1}, \ldots, M_{n}, \rho_{s})
\end{aligned}
\]

(\text{operator})

(\text{operator})

(\text{apply})

(\text{apply})

(\text{predicate})

(\text{if} \ V = \text{false})

(\text{if} \ V \neq \text{false})

(\text{if then})

(\text{capture})

(\text{invoke})

(\text{invoke})

(\delta)

Fig. 2. CEK Machine

The transitions between computational states in Figure 2 deal with the sequential subset of the language that does not involve the store. The transitions extend the CEK-machine transitions [3] by accepting many applications (M_{1} M_{2} \ldots) and abstractions (\lambda \alpha, M) with variable number of arguments. Consequently the continuation code (\text{arg } \rho \ (V_{1}, \ldots, M_{1}, \ldots, V_{n}, \ldots)) conveys the following meaning: the first n components of the application have already been evaluated and their values are V_{1}, \ldots, V_{n}; the next component, i.e., the n + 1th, is being evaluated; the components M_{n+2}, \ldots remain to be evaluated in the environment \rho.

Figures 3 to 6 deal with site transitions. According to rule (sequential), a site can perform a transition if it contains a task that can perform a transition of Figure 2. In rule (fork), the primitive fork creates a new task applying the closure received as argument and returns the new task name.

Figure 4 deals with operations related to boxes. The primitive mkref allocates a new location \alpha, stores the value V in that location, and returns a new box object \text{box } \alpha, V, \text{ref } \alpha, \text{loc } \alpha, \text{pointing at the location } \alpha \text{ in the current site } s. \text{ The primitive defref returns the content of the location associated with the box and the primitive setref changes the content of its associated location.}

Rule (make gp) creates global pointers in the same way as (makeref) creates boxes. A remote service request has the form \text{for str gp M}_{1} \ldots M_{n}, where \text{str} must evaluate to a string naming a handler to be called on the site that \text{gp} is pointing at, with the arguments obtained as values of M_{1}, \ldots, M_{n}. The value returned by a remote service request is the distinguished void value, but its effect is to add a request in the output queue of the current site. A request is not a first-class object, i.e., it is not part of the set of RV values, but it contains the destination site designated by \text{gp}, the serialised string, and the serialised values of M_{1}, \ldots, M_{n}.

Let us observe that the semantics does not specify the behaviour of the Serial and Deserial functions. Their purpose is to convert to and from a suitable format for transportation. The composition of these functions returns a result that is an isomorphic copy of its argument. (The specification is available from [15].)

If there is a request in the input queue of a site, rule (handle) deserialises its content, and applies the handler associated with \text{str} to the value indicated by the global pointer and the deserialised values. It is also worth noticing that the incoming request is handled
\[
\{ \langle C, \gamma \rangle \} \cup T, \theta, s, H, I, O \\
\rightarrow_m \{ \langle C, \gamma \rangle \} \cup T, \theta, s, H, I, O \quad \text{if } C \rightarrow^*_C C_1 \quad \text{(sequential)}
\]
\[
\{ \langle \text{Ret}(\langle \lambda \text{M} \rightarrow \rho \rangle \langle \text{fork}, \theta \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \\
\rightarrow_m \{ \langle \text{Ret}(\langle \tau_1, \gamma \rangle) \cup \{ \langle \text{Ev}(\text{M}, [x \leftarrow \langle \text{init} \rangle], \tau_1) \} \cup T, \theta, s, H, I, O \quad \text{(fork)}
\]
with \( \tau_1 \notin FTN(T) \cup \{ \}
\]

**Fig. 3.** Sequential and Parallel Evaluation

\[
\{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \\
\rightarrow_m \{ \langle \text{Ret}(\langle \text{V} \to \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \quad \text{(makeref)}
\]
\[
\{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{deref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \\
\rightarrow_m \{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{deref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \quad \text{(deref)}
\]
\[
\{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \\
\rightarrow_m \{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \quad \text{(makeref)}
\]

**Fig. 4.** Boxes

\[
\{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \\
\rightarrow_m \{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \quad \text{(makeref)}
\]
\[
\{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \\
\rightarrow_m \{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \quad \text{(makeref)}
\]
\[
\{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \\
\rightarrow_m \{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \quad \text{(makeref)}
\]

**Fig. 5.** Global Pointers and Remote Service Requests

\[
\{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \\
\rightarrow_m \{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \quad \text{(makeref)}
\]
\[
\{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \\
\rightarrow_m \{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \quad \text{(makeref)}
\]
\[
\{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \\
\rightarrow_m \{ \langle \text{Ret}(\langle \text{V}, \langle \text{arg} \rho \langle \text{makeref}, \cdot \rangle \rangle), \gamma \rangle \} \cup T, \theta, s, H, I, O \quad \text{(makeref)}
\]

**Fig. 6.** Mutual Exclusion
anyynchronously and that the handler is executed in a new thread. Optimistic active-messages [25] show that the thread could be created lazily [13] and still produce good performance. According to rule (define handler), the primitive define-handler allows a program to add new handlers in a given site.

Idealised NeXene offers some primitives to implement critical sections. A lock is a datastructure that can be acquired by at most one task using the primitive lock, and that can be released by the primitive unlock. In Figure 6, a busy-wait implementation of locks relies on the fact that transition rules are atomic.

Figure 7 displays transitions between worlds. Rule (site) shows that a world can perform a transition, if there is a site which can perform a transition of Figures 3 to 6. In addition, rules (local request) and (migrate request) deal with request transfers, and rule (remote) is used to create a new world.

Creation of a remote site is always a delicate problem in a distributed system. Frequently, libraries for distribution assume an SPMD approach and use operating-system primitives to startup remote processes. We decided not to model NeXene very closely, as the semantics would have been unsatisfactorily complicated. Instead, Idealised NeXene provides a primitive to create a new remote site and to initiate a computation on that site. This initial computation is essential, because it allows the user to install initial handlers for remote service requests. In addition, it makes it possible to synchronise both sites so that the initial site is kept informed when the new site is initialised and is ready to perform some processing.

A remote site is created by the primitive remote which takes the following arguments: the name of a host, a closure representing the initial computation to be done, and a global pointer that can be used by the remote site to notify the end of initialisation by a remote service request. The value returned by the primitive remote is a global pointer pointing at the new remote site. In order to be implementable in languages without closures, like C, rule (remote) requires the closure to have an empty environment. As a result, the closure can be encoded as a pointer to its entry point. Let us observe that the remote site should be executed on “host”.

Having defined a transition relation between worlds, we can now define an evaluation relation associating programs with values. Detecting termination in a distributed system is a difficult matter in the general case. In Idealised NeXene, we assume there exists a primitive shutdown that the program has to call on the origin site; it terminates the computation and returns its argument.

Definition 1 (Eval) Let M be a closed term. An initial task is of the form \( t_i = \)
\((\text{Ex}(M_{\theta}, \theta, (\text{init})), \tau))\), while a final task is of the form \(t_f = (\text{Ret}(V, \text{arg} \rho (\text{shutdown}, \theta)), \tau)\). We have that:
\[
evalu(M) = V \text{ if } \{(t_f, t_1, s_1, \theta_1, \theta_2) \rightarrow_{\eta} \{(t_f) \cup T, \theta, s_0, H, I, O\}\} \cup W.
\]

\[\square\]

4 The Implementation

4.1 General Description

Figure 8 displays the organisation of the NeXeme implementation. As Nexus is multi-threaded, we had to adopt a thread-safe garbage collector. The public domain Boehm-Weiser's [11] garbage collector supports various OS-level threads and is also part of the PPCR portable common runtime system [26]. On each platform, Nexus is recompiled against the garbage collector (abbreviated gc) and the suitable thread package. The executable is generated by linking the Scheme system with the resulting libraries Nexus, gc, and threads.

<table>
<thead>
<tr>
<th>Utilities</th>
<th>NeXeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>Nexus</td>
</tr>
<tr>
<td>Foreign Interface</td>
<td>Thread Scheme</td>
</tr>
<tr>
<td>GC</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 8.** Organisation of NeXeme

Primitives of the Nexus library are made accessible through a foreign interface definition. Let us note that some Nexus primitives require procedures as arguments (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 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 Section 3. Finally, a library defining a set of utilities provides other paradigms for distribution such as futures, communication channels, or farm processing.

4.2 Distributed Garbage Collection

NeXeme has a distributed garbage collector that takes care of memory management automatically. Our working hypotheses, provided by Nexus, are a reliable message-passing and a FIFO ordering\(^4\) of messages between two sites.

Each site relies on a thread-safe, conservative, mark-and-sweep garbage collector [11]; conservativeness is required as Scheme data are passed to Nexus, written in C, and are pointed by Nexus data structures. In addition, NeXeme maintains two tables for each site. The eexit table\(^5\) associates each global pointer with the number of distinct remote copies of this pointer originating from the site. The entry table contains all global pointers received by a site, except those that point at itself. The exit table, but not the exit table, is a root of the local garbage collector. The role of the distributed collector is to update counters in a safe and consistent way. To this end, it relies on two types of control messages, called "decrement(gp)" and "increment-decrement(gp, s)"), as described below.

\(^4\) Nexus, as opposed to Idelised NeXeme, allows requests to be handled in a non-threaded manner; in such a case, Nexus guarantees the FIFO ordering of requests between two sites.

\(^5\) Different names are found in the literature, such as entry/exit items, scions/stubs [17].
Reference counters are updated according to the diffusion tree [16] of global pointers. The first time a global pointer \( gp \) is serialised, an entry is added into the exit table of the current site with a counter set to 1. Afterwards, for every serialisation, this counter is incremented. Symmetrically, the first time a global pointer \( gp \) is deserialised, it is added to the entry table (if it points at a remote host); if it is already present, NEXeme sends a decrement message for this global pointer, “\( \text{decrement}(gp) \)” to the site that sent the remote service request. When a site receives a message “\( \text{decrement}(gp) \)”, the counter of \( gp \) in the exit table is decremented. Once the counter reaches zero, the global pointer may be removed from the tables only then, the pointer itself and the local data may be reclaimed, if no longer accessible.

In Nexus, a global pointer contains the site it is pointing at, and not the site it is arriving from. Therefore, once a global pointer \( gp \) pointing at a site \( s_2 \) becomes inaccessible on a site \( s_2 \), \( s_2 \) sends a message “\( \text{decrement}(gp) \)” to \( s_1 \). This naive implementation of the reference counter technique is sound if causality is preserved in the system [16]. This is ensured by a reorganisation of the diffusion tree, as explained in the following scenario illustrated in Figure 9.

Let us consider that \( gp \), a global pointer pointing to address \( a \) in \( s_1 \), is migrated to \( s_2 \), and then migrated to \( s_3 \). Figure 9 part 1 shows that the reference counters in \( s_1 \) and \( s_2 \) are equal to 1, meaning that there are \( 1 + 1 = 2 \) active remote references of \( gp \). Once \( s_1 \) deserialises \( gp \), the global pointer received from \( s_2 \), \( s_3 \), a reorganisation can be initiated, as illustrated in Figure 9 part 2. (i) Site \( s_3 \) sends an “\( \text{increment-decrement}(gp,s_2) \)” message to \( s_1 \); (ii) when the message is received by \( s_1 \), the counter for \( gp \) is incremented on \( s_1 \); (iii) afterwards a “\( \text{decrement}(gp) \)” message is sent to \( s_2 \); (iv) when the message is received, the counter for \( gp \) is decremented on \( s_2 \).

![Fig.9. Distributed Reference Counters](image)

Race conditions are avoided between \( s_1 \) and \( s_3 \) by giving priority to “\( \text{increment-decrement} \)” over “\( \text{decrement} \)” messages. As a result, we have the following invariant. For every global pointer \( gp \) pointing to site \( s \):

\[
\sum_{\text{site}} \text{counter}(gp) = \sum_{\text{site}} \text{in}(gp) + \sum_{\text{site}} \text{Value(\text{message for } gp)}
\]

with \( \text{Value(\text{decrement}(gp))} = 1 \) and \( \text{Value(\text{increment-decrement}(gp,s_2))} = 0 \).

The simplicity and portability of the solution is unfortunately counter-balanced by its inability to collect distributed cycles. Our approach differs from “Indirect Reference Counting” [16] because it can reclaim “zombie” pointers, i.e., \( gp \) can be freed on \( s_2 \) even though it remains active on \( s_3 \); other techniques to reduce the length of indirection chains...
are described in [17]. Several optimisations are possible. First, control messages of the
same type may be grouped in a single message. Second, an “increment-decrement(gp, gq)”
to be sent to gq, followed by a “decrement(gp)” to the same destination, may be replaced
by a “decrement(gp)” to gq.

4.3 Utilities
The utilities layer offers a number of services that we briefly describe in this section.
Remote invocation of a procedure sends a locally-available closure to a remote site where
it is invoked on some arguments as in a remote service request, no result is returned by
this operation. Not only does remote invocation of a function transfer and call a function
on some arguments, but also it returns the result to the site that initiated the operation.
Communication channels [21, 9], and futures [12] are also provided.

“Farm-processing” versions of the various remote call utilities are provided that can
be used when a list of global pointers is passed as an argument; these versions map the
corresponding operation onto each element of the list. NeXeme is also able to detect when
a site becomes idle so that lazy task creation [13] and task stealing can be implemented
easily.

5 Discussion and Related Work
The Nexus [7] philosophy is derived from active messages [24], where each message con-
tains at its head the address of a user-level handler executed on message arrival. Nexus
has also been used to build several other distributed languages. In Fortran M [6], program
modules, called processes, communicate via explicitly-declared channels. The language
nPerl [8] is a distributed extension of Perl which provides a remote procedure call; un-
like NeXeme, higher-order functions and continuations are not available. Compositional
C++ [4] supports global pointers that can refer to objects on remote machines, and
defines usual operations on them; also, OC++ offers single assignment objects. Currently,
NeXeme uses Nexus 3.0; Nexus 4.0 provides EZ Nexus a more abstract interface to
remote service requests, which is similar to the one described here.

Boxes allow the programmer to define mutable data structures, for example cyclic
data structures. If a cyclic data structure is passed as an argument to a remote service
request, an isomorphic copy is generated on the remote site. NeXeme does not prevent
the user from mutating each copy separately, as they are now independent entities.
However, using global pointers, any form of consistency can be programmed on top of
NeXeme, such as simulated shared memory [14] or causally-coherent memory [18].

There exist other distributed implementations of functional languages with explicit
notations for parallelism or distribution. DMercury [18] offers a distributed memory model
which enforces coherence based on causality, while ICUSL [19] adds to DMercury trans-
parent remote execution. Kali Scheme [3] also offers a form of active messages, on top of
which higher-level primitives are defined; the semantics of Kali Scheme could easily be
defined in terms of Idealized NeXeme. Distributed versions of SML-style languages are
usually based on communications channels [21, 9].

6 Conclusion
This paper presents NeXeme, a distributed dialect of Scheme, based on remote service
requests and global pointers provided by the library for distribution Nexus. This paper
describes the semantics but also the implementation of this language. The functional
interface to remote service requests of NeXeme is a perfect abstraction to build a Scheme
with futures [14]; in addition, NeXeme is used for programming multimedia distributed
applications.

7 Acknowledgement
Hans Boehm, Carl Hauser, Carl Kesselman, Andreas Kind, Christian Queinnec, Manuel Serrano,
and Steve Thorse gave useful comments and advice for the implementation of NeXeme.
Bibliography


